# A Symbol Table Manipulation Library for the Amsterdam Compiler Kit

by Warwick Heath, October 1988

# Contents.

# Section One.

## 1.0 Introduction.

Prior to detailing the project which has been undertaken this year, it is useful to give a brief introduction to several subjects upon which the project is heavily reliant. These topics are outlined in this section, the section ending in a description of initial aims in undertaking the project. The ensuing report details the various stages of background research, design and development necessary for completion of the project.

Section 2 outlines the background work that was required to gain the in-depth understanding of language issues necessary for the design stages described in sections 3 and 4. Section 3 indicates the choices made in the design of the underlying symbol table structure, section 4 the decisions made in the design of the symbol table manipulation routines.

This is followed in section 5 by a description of the implementation process.

Sections 6 and 7 outline the various testing phases that were used to validate the resulting library, and to test the efficiency of the major routines.

Section 8 summarises the project results.

The Appendices contain the Unix manual pages for the library, a description of the file structure and hierarchy within the symbol table library, and a separate folder containing code listings, example parsers, and tests.

Justification for all design decisions are included within appropriate sections, as are any deviations from the original project aims.

## 1.1 Compiler Symbol tables.

Within all modern programming languages it is necessary to store and retrieve information about the language's user defined symbols. As programming languages have become more complex the amount of information stored has increased, and as the size of programming projects has increased, the number of instances of symbols has increased.

The symbol table is typically a memory resident structure used for the quick

and efficient storage of this information. This storage mechanism is accessed and updated at all stages of the compilation process.

## 1.2 Automatic Compiler Generators.

The compiling of a language into machine executable instructions is a reasonably complex process. However large sections of this process are now understood in considerable detail and can be isolated, breaking the typical compiler into a number of co-operating processes. There are a number of projects currently working on the automatic generation of these compiler components from the language specifications. A number of components within compilers are relatively language independent, the symbol table is one such component, as it's internal operation is totally irrelevant to the rest of the compiler. All that is required, on the most basic level, is for the various parts of the compiler to be able to insert and extract information from the symbol table.

### 1.2.1 The Amsterdam Compiler Kit.

The Amsterdam Compiler Kit (ACK) [TAN80] is based at Vrjie Universiteit in the Netherlands. ACK uses automatic compiler generation techniques as a method for rapidly developing compilers for a wide range of machines.

### 1.2.2 Intermediate languages.

ACK uses the intermediate language technique of splitting the compiler into a machine independent process, the front end, and a machine dependent process, the back end. The front and back ends communicate using a common language, the intermediate language. The front ends of a wide range of language compilers generate the intermediate language from their respective source codes, while the back ends accept the intermediate language as input, generating their respective machine languages from the intermediate language.

The most important result of this is that the writing of a new front end makes a new language available to all machines that have the appropiate back end. This is very different from the alternative approach where one compiler will only execute and produce code for a single machine. Using intermediate languages will produce a wider range of compilers for many different machines,

for less time and resources, than the single compiler/ single machine approach.

There are a number of tradeoffs however. The intermediate language must be expressive, so that it can convey effectively the many different language constructs available in a wide range of languages. The intermediate language must also be easily converted to a wide range of different machine architectures. These may be stack or register based, with varying address modes, byte, word or long. A general intermediate language must support all of the above efficiently.

This in practice is very difficult, in particular the wide range of programming languages precludes an efficient general purpose intermediate language. A general purpose language trades efficiency for generality, a tradeoff which is not suitable for compiler construction. Typically the range of languages and machines is restricted in actual intermediate languages.

### 1.2.3 The EM intermediate language.

The ACK project is based around an intermediate language called EM, designed for block structured Algol-60 type languages and byte addressable machines. The EM language was designed after extensive tests of typical block structured languages.

The overall aim of the ACK project is the rapid development of new compilers for a large number of machines. To facilitate this goal various libraries of routines common to many compiler front ends have been made available to the compiler writer. These are in addition to the more traditional tools, for example parser and scanner generators, that are associated with automatic generation projects.

## 1.3 Project Aim.

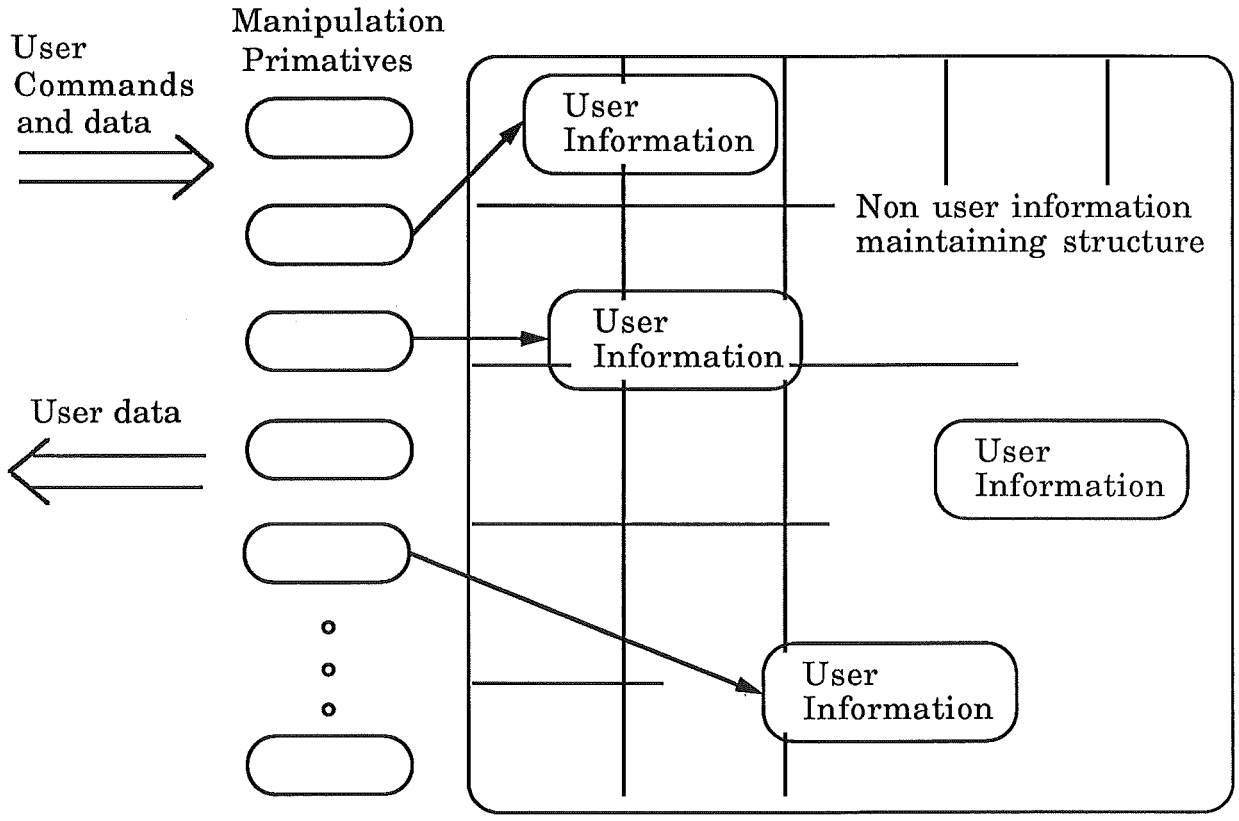
The aim of this research project is to make available a symbol table manipulation module suitable for use within ACK front ends, the module will be able to carry out all the symbol table tasks required for a large range of programming languages. This will free the compiler writer from designing and implementing a structure which is to a large extent common to all compiler front ends. The goal is to provide a module with a standard interface of primitive routines for the interrogation of the underlying symbol table structure. The user need not know the details of this structure, only the various methods of storing

and retrieving their data. It is forseen that this module will be useful not only for the quick production of an experimental compiler but also be efficient enough for production compilers.

An initial step in the project is therefore a thorough and complete examination of symbol table requirements for languages implementable using the Amsterdam Compiler Kit.

**Figure 1.**

Symbol Table Structure.



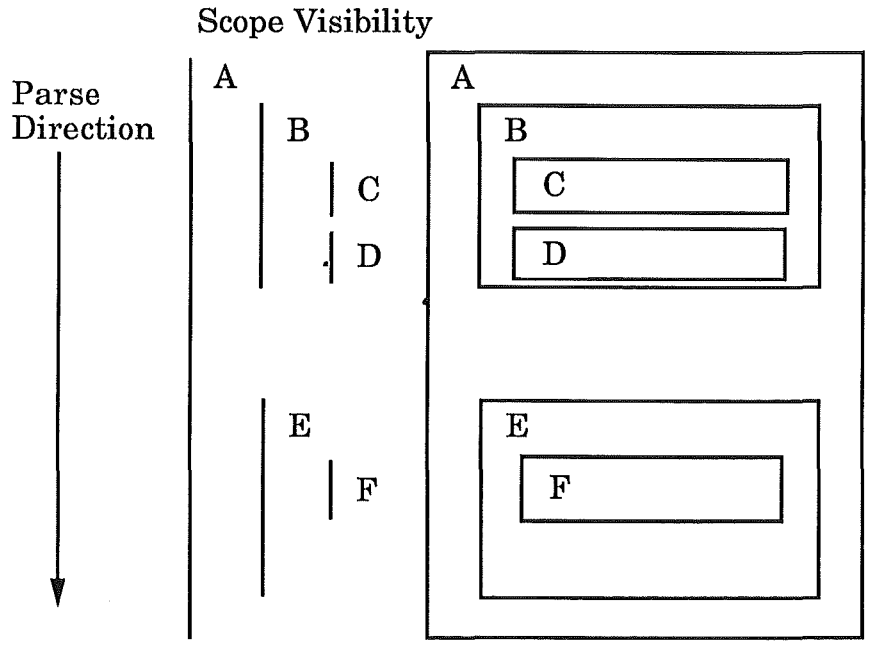
# Section Two.

## 2.0 Background Research.

The Amsterdam Compiler Kit is designed specifically for the implementation of compilers for block structured languages in the Algol-60 tradition. There is now a wide range of languages that conform, to a certain extent, to this heritage.

The development of programming languages has obviously advanced since the day of Algol60, along with the adoption of new language features. Therefore, an examination of typical block structuring was undertaken, along with the important examination of modern extensions and adaptations. This was combined with gaining an overview of the various symbol table structures used and developed over the period since 1960.

**Figure 2 Nested block structure.**



## 2.1 Block structured languages.

Figure 2 is an example of block structure, demonstrating the lexical nesting of block structured languages. Not so clear are the visibility rules that are implicit within block structured languages. The visibility of a symbol is the area of program source where the symbol can be explicitly referenced. Referencing is via the symbol's identifier.

Visibility is an area where languages do tend to differ. Some languages, for example Pascal, dictate that a block may not be referenced before it is parsed, ie the block is not visible. Pascal does have some exceptions to this, most notably in pointer declarations.

Other languages, for example Icon, dictate that all blocks are globally visible, i.e. visible at any point within the program source. These language design decisions were influenced by a number of considerations. In the two cited cases, the only reason that Icon can allow such unrestricted visibility is because of it's restricted implementation of block structure. Icon does not allow nested blocks, and this simplification allows the easy implementation of global visibility for blocks.

Pascal is perhaps more restrictive than is necessary, there is no reason why all blocks at a given lexical level could not be visible within that level.

This example gives a clear indication of the wide range of different ideas and implementations within the basic generalisation of block structured languages. However the general ideas are the same, it is these which are the major influence on symbol table structure.

## 2.2 The implicit hierarchy.

The entire symbol table structure is dictated by the implicit hierarchical scope rules of block structured languages. As each nested block is entered, an entirely new name space is created allowing the redefinition of previously defined symbols. The result is an implicit hierarchy of reference. This is very important as the symbol table must reflect this hierarchy and must automatically extract the most recent lexical definition of any symbol.

The most obvious consequence of this is that the symbol table must be capable

of storing identical symbols, although they may have entirely different attributes. The table must be capable of saving and restoring lexical levels and representing the concept of scope or nested blocks.

Another consequence of typical scope rules is that, at any point within a piece of program text, only symbols defined in an exterior scope and lexically before that point, are visible at that position. The symbol table structure must reflect this also. As previously mentioned there are exceptions to this type of visibility. These exceptions require no additions to the symbol table structure. They are typically handled by forward referencing (an implicit form of the typical Pascal forward declaration) or by multiple parses of the program source.

The data stored in the symbol table is typically cross referenced within itself, for example, a variable symbol is linked to its corresponding type symbol. This type of referencing is language dependent and therefore outside the realms of the general symbol table package that was planned.

## 2.3 Block structured symbol table implementation.

Until this point little has been mentioned of the actual structure of the symbol table. It is now demonstrated that the scope rules lead to a very simple and elegant symbol table organisation, independent of the underlying structure.

### 2.3.1 Opening and closing scopes.

As each new block is encountered while parsing the program source a new scope is entered in the symbol table. How this is achieved depends on the implementation, specific structures are considered in Section 3.

Consequently, on exiting a scope, the corresponding level in the symbol table is discarded. As blocks cannot partially overlap, but must always be entirely within the enclosing block this necessarily means that the most recently created level of the symbol table is that of the current block. This gives a stack based structure to the symbol table, pushing a new level onto the stack at block entry and popping the top level at block exit.

### 2.3.2 Lookup.

This facilitates a very simple search strategy, i.e. if looking for a particular symbol all that is necessary is to search down the stack of symbol table levels. This method is generally the most efficient lookup method, as the most recently declared instance of a symbol is always the currently active one. The most recently declared instance will be the one closest to the top of the stack. Naturally this means that lookup will be most efficient for symbols declared in lexical levels closest to the current one, degrading for references to global symbols. The relative degradation depends, to a certain extent, on the table structure chosen.

This symbol table organisation is therefore typical of all block structured languages, and until recently was all that was required of the symbol table.

## 2.4 Recent additions to block structure.

Ideas about the constitution of program structure have changed considerably since the introduction of Algol60. Many of the revolutionary ideas included in it's design are now being questioned. As programming languages become used in larger and more complex projects, further additions have been seen as necessary to maintain levels of program integrity. These additions are in some cases seen as the logical replacement of block structuring.

### 2.4.1 Contention over value of automatic inheritance.

Over the last 10 to 15 years increasing attention has been paid to the design of languages that allow tighter control of the visibility of their symbols than is available in block structured languages. This has primarily stemmed from work on data encapsulation and its value in the programming environment as a method of modularising program development [HAN81].

The major development in program structure has been the addition to new programming languages of modules. Modules are blocks which require the definition of an explicit interface with other blocks. This means that there is no automatic inheritance from exterior scopes. Modules are closed scopes, and can be treated as separate units. This explicit interface can be made incorruptible at compile time, the various advantages of this feature are discussed in the next section. There have been many arguments recently in favour of an explicit

interface, in preference to the implicit one typical of traditional block structuring.

### 2.4.2 Block structure verses Modules.

There are a number of recent articles [HAN81], [CLA80] criticising block structure and advocating modules as the new basic unit of programming languages. The arguments include,

-the greater legibility of module structure, (deep nesting tends to separate block headings from their bodies);
-the ease of separate compilation with an explicit interface;
-the ability to explicitly state the call structure of a program;
-hiding a module's internal structure.

These arguments however are not an indictment of block structure, but rather one of automatic inheritance. Perhaps the most sensible solution would be to require nested blocks to explicitly import external symbols that they reference. This is, for example, the approach of Euclid [POP78], a modern language designed with the goal of being able to carry out program verification. The language's designers obviously saw implicit inheritance as an unsafe aspect of language design, I believe that this is a significant point against implicit inheritance.

The approach taken by other modern languages has tended to be a less radical departure from implicit inheritance. What we have therefore is the same general concept but many different implementations.

Modula-2 for example has three different types of modules - all of which require different treatment, and have different scoping rules.

## 2.5 Difficulties for the symbol table.

Modules present some difficulties to the maintenance of the symbol table. These difficulties are not because of any inherent problems within modules themselves, but rather because of the fundamental differences between the pure module concept and nested block structure. By pure module structure I am excluding the nested module constructs that are available in, for example, Modula2 [WIR83]. Nested modules appear to me to be an anomaly. Nesting is used to restrict access to a module whereas the module's explicit interface can be used to achieve this. The only advantage that I can perceive is, if, only the enclosing

module is ever going to access the nested module.

### 2.5.1 Varying visibility.

These fundamental differences revolve around the visibility of symbols within program source. Whereas in the simple block structured case all symbols were visible according to a simple set of scope rules, with modules we have symbols moving in and out of scopes regardless of their original lexical level. This is in contrast to the traditional stack structure of a symbol table.

We require a consistent method to combine these two methodologies, retaining a stack based table but adding facilities for modules. Modules are created as closed scopes i.e. when searching within them we do not automatically search enclosing scopes for symbols that cannot be found. This isolates the lookup process allowing modules to be incorporated in the stack of scopes. All that is required is to stop searching at closed scopes when moving down the stack.

### 2.5.2 Implementation of Imports and Exports.

It remains to construct a consistent method for handling imports and exports. Imports and exports are the commonly applied term to those symbols that are specified as being able to cross module boundaries, through the interface. Imports are symbols from other modules used within the importing module, exports are symbols from the exporting module used within other modules. Note that import and export are a mutual undertaking, a module can only import those symbols from another module that the module allows to be exported.

Similarly symbols will only be exported if another module specifically imports them. Naturally there are exceptions, for example in Modula2, modules that contain nested modules implicitly import all of their nested module's exports. Example 2.1 shows the typical module interface, using a trivial example from Modula2. Note that the two modules are in separate files.

```
DEFINITION MODULE simple;
      CONST
            happy = 0;
            sad = 1;
            (* This module can export 'happy' and 'sad',
               no symbols are imported *)
END simple.
MODULE main;
FROM simple IMPORT happy;
      CONST
            angry = 10;
      VAR
            DrSeuss : CARDINAL;
BEGIN
      (* 'happy' and 'angry' are visible here, however
         'sad' is not because it was not imported. *)
      DrSeuss := happy;
END main.
```

What is required, is a method for implementing these imports and exports between different scopes. Also, it must be noted that these different scopes are not necessarily resident in the symbol table, they may be separately compiled units. If this is the case then symbols may have to be loaded from secondary storage, or perhaps from a memory resident structure used for storing saved module symbol tables.

The imported or exported nodes could be physically moved into the new scopes. This has the advantage that additional storage is not required. There is some overhead involved, as this would require the unlinking of the symbol from the structure, and it's subsequent relinking in another position.

The other main option is to create a new symbol in the current scope and link this back to its actual definition. This has advantages in that there is less overhead involved at scope entry and exit. The symbol can be created and deallocated at the position it is used.

A similar idea is to copy the imported/ exported symbol into its new scope. Again, there are various overheads involved which must be assessed and compared with reduction in lookup time. These overheads involve the time required to copy data, compared with delinking and relinking symbols in a different location. In reality these overheads are likely to be small, and will only be incurred a few times for each imported/exported symbol.

At this stage a decision about a likely course of action had to be made. The method used in incorporating modules within the table was a major design decision.

The method chosen in the library under development was to give the user the

opportunity to choose whether a complete copy was to be made or whether to link the imported symbol back to its actual definition. Moving the actual nodes within the symbol table was rejected as being too high an overhead on scope entry and exit. More importantly, it requires the user to know that the symbol is being shifted within the structure, and that it's position is changing. This is obviously not desirable. As there is no standard way of handling modules, either the manipulation module must make assumptions about reinstating imports/exports at scope exit or the user must tell the symbol table module what to do.

In the first case the module loses its generality and extra information has to be stored. This is a situation which should be avoided by a general structure such as the one which is being designed. Any arbitrary decision about the implementation of modules will decrease the versatility and range of languages supportable. For those languages that are able to be supported, there will be the danger of a large amount of redundant data stored. The data is redundant because the symbol table structure would have to store it, to keep control of exporting and importing symbols. The user would also typically have to store exactly the same information for use during the compilation process. Redundancy of this nature is not desirable for two reasons. Firstly, the extra storage overhead may be significant during the compilation process, and secondly extra time will be spent copying and transferring unnecessary data.

If the user must tell the symbol table module what to do, the user must be aware of the underlying symbol table structure changes that delinking and relinking a symbol entails. The user must specifically reinstate the symbols to their original locations when they are no longer required in the scope to which they were moved. This belies the entire idea of building a generic symbol table module. The user should not have to be aware of this detail.

### 2.5.3 External compilation.

The other major difference with modules is that they are specifically designed to be externally compiled units. This requires the saving and relinking of scopes, depending on import/export lists. To a certain extent this sort of facility is necessary even in simple block structured languages, both for separate compilation and as a facility for storing symbol table information in multipass compilers.

This facility can be provided with relative ease. Naturally there are only certain conditions when such relinking can occur. This however must be left to the

user to control as, again, it is language dependent.

## 2.6 Effect on symbol table design.

The overall effect of modules on the structure of the traditional table structure is therefore manageable. What is actually required is a number of symbol table routines which the user can use to implement modules within the traditional symbol table structure. This is the method chosen by the ACK Modula2 compiler for example.

While looking at alterations to accommodate modules, other symbol table structures were considered, but my resultant designs were restricted in usefulness to languages based on modules. The performance of my designs would have been poor for the various combinations of modules, nested modules and modules with block structure prevalent in contemporary languages.

The designs were loosely based around the idea of modular symbol tables, separate sub-tables for each module with the global symbol table being constructed from module sub-tables. This method rapidly becomes complex when combined with traditional symbol table structure. For example how should the implicit hierarchy of normal Algol60 scopes be represented? The answers are not intuitively obvious. This is just one of a number of problems that were uncovered.

Investigations along these lines were therefore abandoned in favour of the traditional structure -i.e. scope/block levels within the symbol table. This structure requires only minor alterations to be able to effectively incorporate modules.

# Section Three.

## 3.0 Design of symbol table structure.

This section gives a rationale for the structure decided upon for the generic symbol table, and the information stored to maintain the structure and give it the required functionality. This section progresses from initial thoughts, through to general structures considered, and their relative merits. The section concludes with a description of the final structure.

## 3.1 Initial thoughts.

On first approaching this project, the emphasis seemed to be on a generic symbol table module, one that would handle all the facilities provided by the range of programming languages under consideration. A first attempt at design therefore placed a large emphasis on the data which would have to be stored to maintain various features, for example imports and exports.

### 3.1.1 Examining existing compilers.

To this end, some time was spent examining the ACK Modula2 compiler, seeing how various features were handled within the table structure that was used. A list of language features and strategies for implementation were compiled, mainly taken from the ACK Modula2 and C compilers, and the papers [GRA79] and [REI83]. I considered adding information to store details about such features, however it was rapidly discovered that what was actually required was a robust basic structure with the minimal amount of information stored.

Any additional information stored, above that which is necessary to maintain the structure, is potentially redundant. The user can only access their own information stored in the table and so may have to replicate information stored by the structure. The only information that the structure must store is that required by insertion and lookup, which in a symbol table is the symbols identifier, a text string.

The symbol table's structure and information must be irrelevant to the user. This is the most important consideration in a general module. The facilities must be provided to allow the user to do anything desired. Conversely anything that places a restriction upon the user must be avoided. The user must therefore accept a certain responsibility to gain this flexibility, so must keep track of symbols with special attributes and use the primitive routines to manipulate the structure.

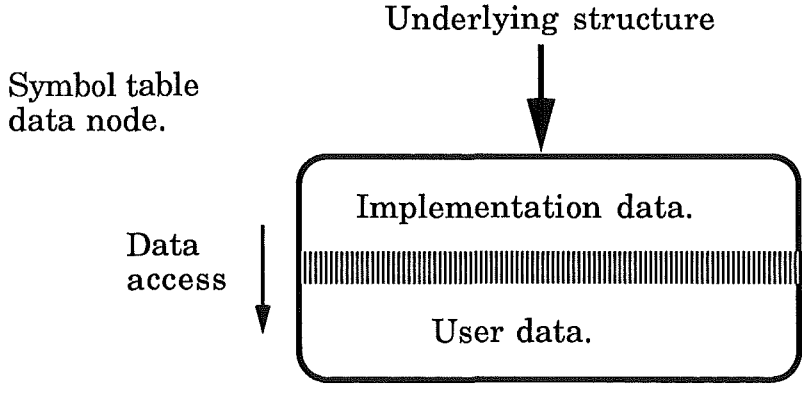
### 3.1.2 Clarification of objectives.

At this stage the general aim of the project was clarified and objectives set. A general, internally maintained structure would be presented to the user. Within this, the user could store, retrieve and manipulate symbols using a variety of provided routines. Facilities would be provided (by way of options), to allow the user to control various aspects of the operation of the symbol table. A typical example of this would be the size of the symbol table.

## 3.2 Design.

After making these decisions it was the next step to finally look at the internal implementation issues. All work is based around the observation that each node representing a symbol in the table must contain a user defined area of storage and an area of storage associated with the symbol table implementation. Figure 3.1 demonstrates this characteristic. The only method of access is via routines provided by the symbol table.

**Figure 3.1 Components of an individual symbol**



### 3.2.1 Available structures.

These nodes must then be stored within a traditional symbol table structure. Because of the necessity for efficiency there are only two possible organisational methods available, a hashed structure or a tree structure. There are a number of types of each of these methods. Trees are classified according to their degree and whether they are balanced. Degree is a measure of the number of nodes which emanate from a single node. A balanced tree has an even structure, any paths from the root node to a leaf node will be of length N or N-1 where

**D*N >= number of nodes in tree > D*(N-1)**

and D is the degree of the tree. There are many variations of trees, in use they differ in their operational characteristics. Traversal (equivalent to the lookup process in a symbol table) is generally faster in complicated trees, however insertion is generally much more complicated. In practice binary trees are very effective, giving rapid retrieval and being simple to construct.

Hashed structures all rely on a common principle, the difference is the method of collision resolution. The only practical collision resolution scheme for a symbol table are linked list chains. Linked list chains allow the maintenance of the symbol table stack structure, the collision chains can be regarded as a stack, the closest symbol to the hash bucket is the top of stack.

The relative merits of unbalanced binary trees and hashed tables with collision chains were therefore studied in greater detail.

The search method used for each of these methods is relatively straightforward. Both methods can easily and efficiently reflect scope rules and block structure. There are various advantages associated with each of these methods.

### 3.2.2 Binary Trees.

By using a forest of binary trees, one tree for each scope level, very simple insert and lookup routines can be implemented. Insertion is the traversing of the tree structure until the appropriate leaf is found. Lookup requires tree traversals starting at the current tree, until the identifier is found. Insert time is dependent upon $\log_2 n$ where $n$ is the number of identifiers in the current level. Lookup is similarly bounded.

This is however average case behaviour. As the tree is not balanced it cannot

be guaranteed that a tree structure will result from any given sequence of insertions. In the worst case, insertion and lookup may become O(N) as a linear list from the root node may result. This may not happen often but certain programming practices may cause a rapid degeneration of symbol table performance. An example of this is programmers who order their declarations alphabetically or program generators that generate alphabetic sequences of identifiers.

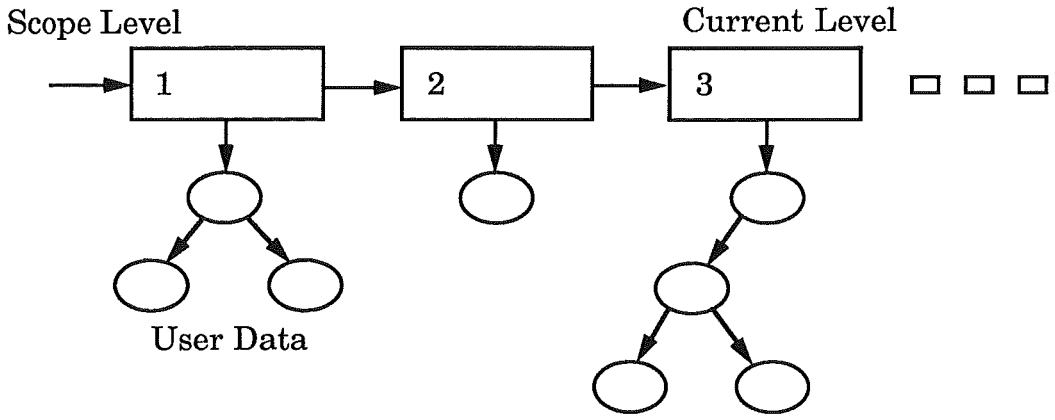
This problem can be redressed by balancing the binary tree. Some extra overhead is involved but to consider binary trees as a viable alternative they must be balanced. Balancing is a cost which is only incurred once for each symbol, on insertion, so the cost is deterministic.

Scope entry is straight forward, a new tree is added to a vector of scope trees, similarly scope exit requires the deletion of a tree from the vector.

Another consideration is the storage space required. Because the structure is built incrementally there is no storage allocated that is not in use. The storage space required to maintain the structure is minimal, two pointers per node. Binary trees are therefore very efficient in their use of memory. This is no longer a major consideration when constructing a storage structure as modern virtual memory systems provide the user with large address spaces. A degradation in response time, due to a large structure, is the only concern.

Figure 3.2 illustrates the use of a vector of unbalanced binary trees to represent scopes.

**Figure 3.2**
Binary Tree Forest.



### 3.2.3 Hash tables.

Hashing is the process where an arbitrary number of characters are mapped to a number within certain specified bounds. The bounds and mapping depend on the function used. Generally it is desirable to find a function which will distribute any given sequence of strings evenly over the desired range.

Hashed symbol tables consist of a hash addressed vector where the hash key is calculated from the symbol identifier. Symbols are stored in the vector at the location indicated by their hash value, collisions are handled by sequential chains from the hash bucket. Insertion is always O(1) as insertion always takes place at the front of the collision chain.

This is more effective than the binary tree, however the bounds on lookup can be much worse than binary. Lookup involves searching the collision chain of the bucket that the identifier hashes to. If the hash function is distributing the identifiers evenly the collision chains will be on average of (N/V) length, where N is the number of identifiers in the table and V is the number of hash buckets in the vector. In practice however, good hash functions are difficult to find, there are no algorithms to produce good hash functions. All that exist are certain heuristics which may be followed, generally to avoid poor hash functions.

The worst case is that the hash function will map every identifier to the same hash location. In this case lookup is order(N) where N is the number of symbols in the table. This is not desirable, however with sufficient time and testing, a function can generally be found that is very unlikely to degenerate in this way. The hash function is less likely to degenerate to it's worst case than an unbalanced binary tree.

The best case is when the identifier being sought is at the beginning of the collision chain. Generally with a load factor of less than 100%, (N less than V), lookup will on average approach best case and be O(1).

Scope entry requires the addition of a new level to a scope level vector, and all nodes within the same scope must be chained together to allow rapid delinking from the table on exit. This is potentially the largest overhead in a hashed symbol table, the iteration through all the nodes in the current scope on block exit.

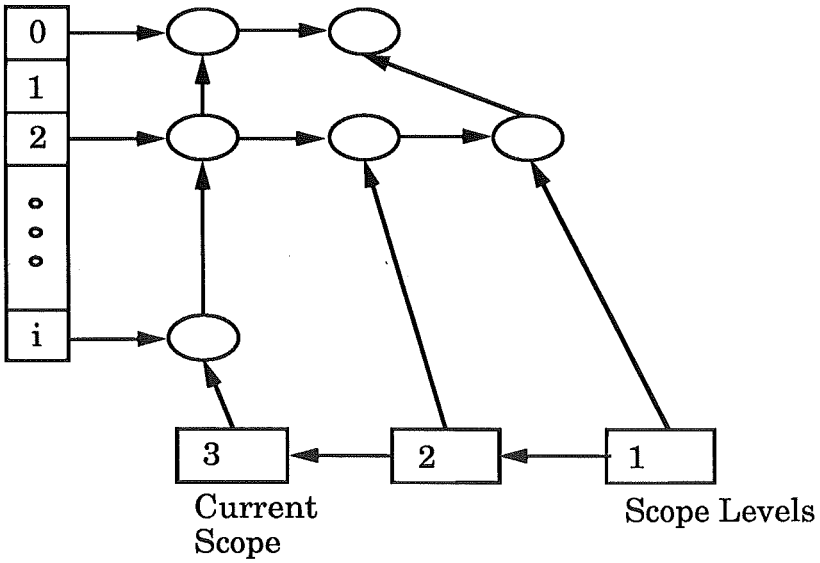
The storage overhead for hash tables is greater than binary trees. No matter how many identifiers there are in a table there will always be the entire hash vector resident in memory. This could be overcome by creating the hash vector as a linked list, but we then lose the major advantages of a hash structure. This advantage is the ability to access a bucket in order(1) time, by a direct array access.

As mentioned earlier however this space overhead is no longer as important as it used to be, the overhead involved in this case would only be of the order of several thousand bytes.

Figure 3.3 shows a hash table using collision chains to represent the scope stack.

**Figure 3.3**

Hash Table.




### 3.2.4 Decision on basic structure.

A hashed symbol table was chosen for three major reasons. The most important advantage is the potential efficiency i.e. rapid insert and lookup. Lookup is particularly important in symbol tables, the advantages of O(1) average lookup time for an unloaded table is important. The major difficulty is to construct a suitable hash function.

The other two reasons are principally pragmatic ones.

Firstly there is an example of a hashed symbol table already included within the Amsterdam Compiler Kit. This was originally used in the ACK C compiler and was separated out to form a very simple mechanism for building symbol tables. All that is provided is the hash structure and basic memory allocation primitives for the structures. I will go into the mechanics of this in greater detail in Section 7.

Secondly, a paper [GRA79] details the implementation of explicit scope control within hashed symbol tables. This is shown to be relatively simple as the linear list resolution chains can be easily manipulated to allow the import and export of symbols. This is described as being the process of moving a symbol's position within a chain to reflect its lookup precedence. This movement can be effected by copying the symbol or by referencing the original, as detailed in Section 2.

This was important as it gave confidence that the structure would be able to support modules. The major feature of the system is that it cannot be implemented within binary trees. This does not mean that binary trees are unsuitable, just that a comprehensive system for implementation of imports and exports was not uncovered during the research undertaken.

### 3.2.5 Support structure.

Once the basic structure of a hashed symbol table had been selected, it was simply a case of determining the support structure, i.e. the vector for scope level and the various linkages required between the nodes in the table. It was also decided that there must be a facility for saving entire scopes within the symbol table. This is needed for multipass compilers for example. Uses of saved scopes are much more widespread, modules are merely closed scopes and so can be saved as they are exited. Symbols can then be exported from the saved module later in the compilation process. This may be at a later date, the scopes having been saved to secondary storage.

The table was designed as C structures linked by pointers, as discussed in the following outline and augmented by Figures 3.4 and 3.5.

It was decided to implement the symbol table as a node containing;

-the hash table (an array of pointers to symbols),

-pointers to the external scope node and current scope node,

-and an array of pointers to saved scopes.

The symbols are nodes containing:

-a pointer to the next symbol in the hash chain,

-a pointer to the next symbol in it's scope,

-and a pointer to its scope node.

The symbol node also contains the user defined record into which the user stores information. The symbol node must also store the identifier which is used as the hash key, this is the only information that needs to be stored.

### 3.2.6 User hooks.

However as an important feature of the table, an additional integer is stored for use as a wildcard. Many symbol table routines have been designed to use this wildcard or user 'hook' to allow the user to select groups of symbols from the table. For example at scope exit, the user may have to process all the nodes in the scope.

Therefore a method must be provided for the iteration through all of the symbols in a scope. To save the examination of all symbols the user can look for specific types of symbols by iterating over only those nodes that match the wild card. Many symbol table functions iterate over selected nodes, matching a wild card parameter with that stored by the user in the node.

The final node is the scope node, this contains;

-links to the immediately enclosing and nested scope,

-a link to the first symbol in the scope

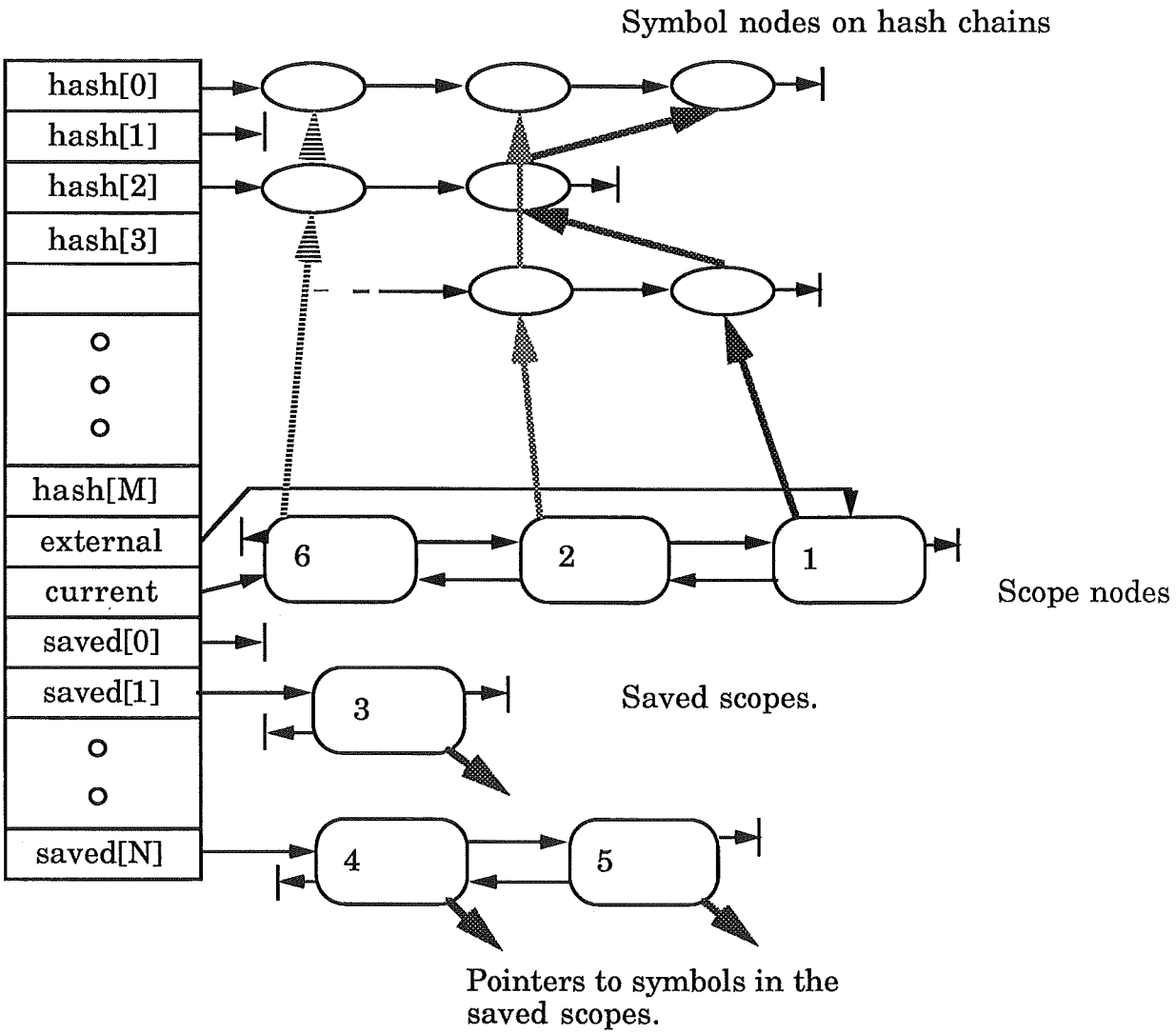
-and a scope identifier.

The scope identifier is so that scopes can be identified by name. This is necessary in some languages, for example Ada, where a variable reference can be referred to by its scope name. This is similar in principle to the scope widening in structured data types such as Pascal records or C structures. As with these types, provision must be provided to reference scope variables in the same way as record or structure fields are referenced.

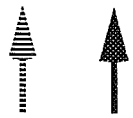
Figure 3.4 shows a slightly simplified view of the symbol table. Omitted are the links from each symbol node to it's scope node.

Figure 3.5 gives a more detailed view of the symbol node, expanding on Figure 3.1.

**Figure 3.4 Diagram of symbol table structure.**

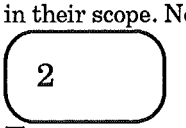



**Key.**



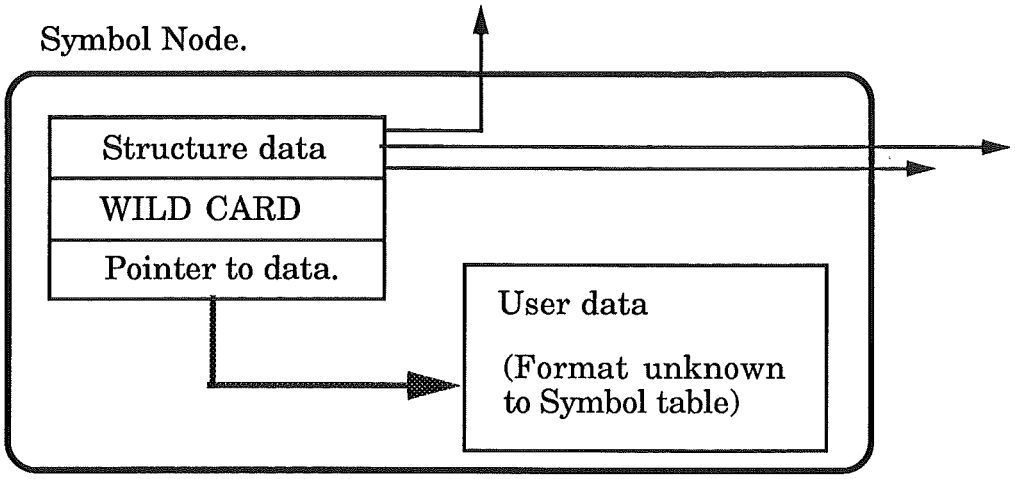
These type of arrows indicate the linkages between each symbol within a scope. Not shown in the above figure are the links from each symbol node back to their scope node. Each of the saved scopes also have a list of delinked symbols, these are also indicated by the above arrows.

These are symbols stored within the structure. Each one is linked into a hash chain and to the next symbol in their scope. Not shown is the link back to their scope node.




These are scope nodes, the number being an identifier, as each scope can be referred to by name. There are two types of scope nodes in the table, those that are currently in the symbol table, and those that are saved.

**Figure 3.5 Symbol Node.**




**Key.**

This is the pointer which is the only method for accessing the user data.

These indicate pointers used to maintain the structure of the symbol table.

# Section Four.

## 4.0 Design of Library routines.

This section describes various routines that were needed to give the symbol table module the required functionality. The section starts by describing the routines that provide the basic features of the symbol table, moving on to discuss the routines that implement the more advanced features.

As the various features are discussed, the names of some of the routines used to implement them will be mentioned. To gain a comprehensive view of the range of library routines, reference should be made to Appendix A which contains the Unix manual entry for the library.

## 4.1 Basic Features.

The basic features needed in any block structured symbol table are opening and closing scopes, insertion and lookup of symbols. The operations are typically in themselves simple, however because the symbol table library has to operate over a wide range of languages, these operations have become more complex.

### 4.1.1 Opening and closing scopes.

Levels within the symbol table must be opened and closed as scopes are encountered during the parsing process. At the lowest level this only requires the linking into the symbol table of a new scope level. Because of the requirement in some languages that scopes be referred to by name, these levels may be named, but the facility must also be provided for anonymous scopes. `Open_scope()` and `close_scope()` provide these two features.

The concept of open and closed scopes must also be provided. Open scopes allow unrestricted access to symbols in external scopes, closed scopes corresponding to modules or other language features that do not allow the automatic referencing of external symbols.

### 4.1.2 Lookup and insertion.

`Lookup()` and `insert_wild()` implement the other two basic features.

These routines must be flexible enough to be able to accommodate all other features, as all of the complex features use these basic routines. Insertion is simple because of the hash table structure. The user must only specify the table and symbol and the user wildcard at insertion. The `insert_wild()` routine returns a pointer to the user storage for the newly created symbol. Lookup is the most complicated process. `Lookup()` must be able to search in specified scopes, keep track of open and closed scopes, terminate searches correctly, and keep track of previous searches in the case of multiple lookups. Each of these features will be discussed in more detail in this section. The danger with such a complicated lookup routine is that it will become inefficient for most lookups. This is a problem as lookup is the most common operation on a symbol table, it must be as efficient as possible.

## 4.2 Complex Features.

At this point my initial research was used to decide on a collection of routines which would effectively cover the diverse range of language features being considered. Below are the major facilities it was considered necessary to provide, many end up using the primitive routines. This is a tradeoff between speed and building a cohesive library. There would be advantages, in execution speed, in having many different lookup routines, one for each different complex operation. This was decided against, as being contrary to the library approach that was being undertaken. The efficiency of the routines was studied in light of this decision, the results are in Section 7.

Some of the following features have already been briefly discussed.

### 4.2.1 Iteration.

In the simple case it is often necessary to iterate over every symbol within a scope performing some function. A common example of this is at scope exit, where the compiler may check that each symbol has been referenced at least once. In a user maintained structure this process is simple, it is merely a case of iterating through a linked list of symbols. This cannot be done by the user in the library being designed as it would presuppose some knowledge of the structure.

In a more general case, it may be required to iterate over only certain types of

symbols. This requires that the symbol table knows something about what is stored within the user data. This is where the user 'hook' or wildcard, stored in the symbol table structure is used. The iterator is designed so that it matches only those symbols which have a certain value in the wildcard. This has been implemented in a general way so that any single value can be matched, certain groups of values, or every symbol in a scope may be matched.

These iteration facilities are provided by two routines, `iterate_start()` initialises the iterator with the scope to iterate over, and the wildcards to select. `Sym_iterate()` may then be called successively, each time returning a user symbol, until the scope has been exhausted.

Iterations are only as useful as the wildcard facility, so routines are provided to reset wildcards,this may be specified in terms of the symbol's previous wildcard value.

### 4.2.2 Delinking and Restoring Scopes.

This is a fundamental principle of the symbol table library. A scope is a level of the symbol table that has special attributes. The scope is increasingly becoming a separate unit within programming languages, therefore special facilities must be provided for the saving of these scopes. As previously mentioned, the scopes have a separate hash table, similar to the symbol hash table.

Routines were designed to allow the rapid search for, and relink of, a scope. Relinking consists of making the relinked scope the current, or most nested scope in a certain table. Examples of these routines are `widen_scope()` and `restore_scope()`. Scopes can be automatically saved on exit, and can be referenced by their name or by a pointer to them. Reference by pointer is provided so that the user can store a saved scope within their own data. The most obvious use for this is for Pascal record or C structure types. The fields of these types are typically a separate scope which can be saved within the user data. The scope can then be relinked into the table using the symbol table routines.

### 4.2.3 Lookup in Named scopes.

In languages such as Ada, symbol references may be qualified by their scope name. Qualification is necessary because symbol names may be overloaded or in multiple use within a single scope. In normal block structured languages this would mask references to any external symbol of the same name. To implement the Ada qualified variable, the symbol table must be able to commence lookup in

scopes other than the current one. This leads to another essential facility, the ability to perform multiple lookups.

### 4.2.4 Multiple Lookup.

As another consequence of the aforementioned overloading, a method must be provided to lookup all occurrences of a single name within a scope. This is necessary because some languages use context to determine the current binding of a name. Iteration is therefore provided on names, with a multiple lookup facility. This iteration over names is similar to that provided by the symbol iteration routines described earlier in the section. Most of the multiple lookup facilities are provided by `multiple_lookup_in()`.

### 4.2.5 Storing Scopes.

As a prelude to separate compilation and modules, the issue of saving scopes and reloading them to and from secondary storage must be addressed. It was decided that ASCII text would be the best way to store data, for the simple reason that it is portable. The major problem with saving and loading is that the symbol table routines must save and load the user data. The format of the user data is unknown.

To actually copy the data to disk would be relatively simple however pointers within the user data would be lost and the user would have no method of relinking these pointers. Therefore the user must save their own data. The method chosen was to have the user define routines which the symbol table routines could call whenever a user record was to be written or loaded. The symbol table routines would pass the user data and a file pointer to write to.

The user must construct a systematic way of loading and saving data. There will be cases where the user will not be able to fully reconstruct a symbol. This could be a result of the symbol using a pointer field to reference a symbol that has not yet been loaded. This requires some saving and backtracking to finish relinking loaded symbols, as they are being loaded. This is encountered in any situation where data structures containing pointers are saved to secondary storage.

This method, using the routines `save_scope()` and `read_scope()`, has worked well in practice.

### 4.2.6 Import and Export.

These are specific applications of a general feature that has been provided. It was decided to allow the arbitrary movement of symbols through the symbol table. The wildcard is used for selection and the selected symbols may be copied to, or referenced from, the current scope. Imports and Exports are therefore implemented by using a wildcard to selectively import only those symbols that match either of these values. Facilities were designed to allow the movement of entire scopes or just selected symbols from scopes. These scopes can be referenced by either the scopes name (i.e. module name) or by scope pointer. This latter method is provided for use by nested modules. Once a module scope has been closed it is not necessary to keep the module name to reference its exports, only a pointer to it. This would be useful in languages such as Euclid where modules are types, as variables typically contain pointers to their types within the symbol table.

Modules can also be used to store language globals, so that the symbol table can be preloaded from secondary storage. An example of this is included in the example Pascal parser which demonstrates the use of the symbol table library. Symbols such as integer, char and writeln are loaded from a module into the global scope.

### 4.2.7 Debugging.

As, within any system, debug information is useful, routines have been designed to allow the symbol table to be dumped in a variety of ways. The table can be dumped showing the current scope structure or the symbols stored in the saved scopes can be dumped. The information may be written to standard output or to a specified file. Facilities are also provided to take 'snapshots', these write to successive numbered files that part of the table that was most recently dumped. Facilities such as this are obviously necessary while compiler building, and are provided by the routines `dump_table()` and `snap_shot()`.

### 4.2.8 Disposal.

A variety of disposal routines are provided making use of the ACK memory allocation library. Care must be taken on deallocation when using imports and exports as one option for implementing imports is to reference the original copy. These original symbols must not be deallocated while there are still references to them. The library automatically keeps track of symbols that are referenced more than once, so that these symbols may not be deallocated.

## 4.3 A simple aggregate function.

The main idea with designing the library was to provide primative routines from which a wide range of more complex routines could be built by the user. A simple aggregate which has been included is a local insertion routine. This is included because it is probably the most common routine that will be executed, it is included as an example of how it is forseen users will construct their own aggregates. The routine first checks to make sure that a symbol with the same name does not already exist in the current scope - if not the symbol is inserted.

## 4.4 Multiple and default tables.

Various languages need multiple symbol tables. C for example, requires a separate symbol table to store structure and union names. Obviously facilities must be provided to specify the appropriate symbol table when calling symbol table routines. This is not necessary in most cases however - many languages only require one symbol table and those that have multiple symbol tables frequently use only one for most lookups. Most common routines can therefore be accessed through the provided aliased function calls that take a default value for the symbol table. The user can specify which table is the default. The aliased function calls are accomplished using the C preprocessor, as is the setting of the default table. There are also a number of aliased routines which assume the current (or most nested) scope.

# Section Five.

## 5.0 Implementation.

This section gives a brief outline of the various coding and implementation phases that were encountered during implementation of the symbol table library. It progresses from the coding of the basic routines through to an introduction to testing that was carried out at the end of the library development.

## 5.1 Initial Coding.

The background work and design of structure occupied the first two months of work. Before starting coding it was necessary to become familiar with the libraries that already existed in the ACK library tree. There is a memory allocation module available which takes care of heap memory allocation and deallocation under C. This was the only library which it was necessary to use. Finding the object files and linking them into some test programs provided some experience of how the ACK modules function.

A slight learning curve was encountered here as various aspects of programming in C under Unix were learned. This included such things as designing makefiles, searching for and linking remote libraries and learning C.

The four basic routines were coded first, open and close scopes, insertion and lookup. It was decided to start by implementing the basic functionality first. The code for the more complex features was not written straight away, the approach was to write and test the code in stages. Foremost in my mind has been the knowledge that to be a library package, the code must be thoroughly tested.

The initial code allowed the creation of a symbol table, opening and closing of scopes and simple hierarchical insertion and lookup. Even at this stage there are languages for which this would be useful, for example the Bach programming language implemented by Rod Harries [ROD88].

## 5.2 Preliminary testing.

This initial version of the library was linked into an assignment that was being worked on at that stage. This assignment was the implementation of semantic checking within the Bach programming language, and as such required a symbol table. The symbol table requirements of Bach are extremely simple, and were easily handled by the symbol table library. However there were problems with bugs within the symbol table, and also it was not very easy to link it into the assignment as an adequate file structure had not been worked out at that stage. Because of these difficulties, mainly the former, the use of the library was discontinued.

The testing procedure here was fairly laborious, merely consisting of setting up programs which made calls to symbol table routines. The debug routines were used to verify that the desired results were being obtained. This is a difficult method of testing, as the sequence of symbol table calls must be carefully chosen so that an illegal sequence of calls does not occur. An example would be closing a scope before any had been opened. In a real implementation of a compiler this error would be detected by the parser, the symbol table would not have to handle it. This is a simple example, more subtle ones caused some unnecessary debugging.

## 5.3 Further implementation.

The next step was writing the rest of the routines, and to continue the debugging process. The project writeup was started during a break from debugging and testing the routines.

While testing the second set of routines it became obvious that more systematic methods of testing were needed. Up till that point the project had to be relinked for each different test. This was as a result of having a number of main() programs consisting merely of symbol table calls. To improve this, a general test shell was written. It is a simple program which reads symbol table commands from a command file (in a rudimentary text format) and executes them. This enabled the definition of a number of scripts testing features singly and in combinations.

## 5.4 Library structure and final testing.

The project was implemented as a number of separate files as is typical of C. C provides some basic module-like facilities, emphasising separate compilation of files. These facilities were exploited during development, the resulting file structure is described in Appendix B. The ease with which this could be accomplished in C is important as it allows the symbol table to be linked into a number of different applications easily.

The major part of the implementation appeared to be complete at this point, obviously extra features were added as their usefulness became apparent. The emphasis now switched to the twin goals of testing and proving the versatility of the library. It was decided to use a Pascal parser to test these attributes. Pascal was chosen because of its typical block structured approach to scoping. Perhaps more importantly it is a subset of most modern languages and testing Pascal gives results that can be generalised to a wide range of similar languages. Pascal is typical of the languages that the Amsterdam Compiler Kit's intermediate language (EM) was designed for. The implementation of the Pascal Parser is detailed in Section 6.

This successfully tested the basic aspects of the library. However a large number were still untested in a real application, specifically routines dealing with modules. It was decided to extend the Pascal parser to incorporate modules. Wirth's grammar for Modula2 was used to alter the Pascal grammar of the Pascal parser. The success of this testing phase is also described in Section 6.

There remained one question regarding the library, a question which is quite critical to the useability of it, that is it's efficiency. To a large extent the efficiency of a symbol table is reliant on it's lookup method. This is because it is the activity which is typically performed the most often in symbol table operation. In hashed symbol tables this lookup time is mainly influenced by the efficiency of the hash function. If the hash function is poor many identifiers may hash to the same location, causing long linear searches along collision resolution chains. It was therefore decided to test the hash function to ascertain how good the spread of identifiers were. The testing of the hash function also gave a good opportunity to test the speed of various symbol table routines. The results are outlined in Section 7.

# Section Six.

## 6.0 Testing the Symbol table library.

An important part of the project has been the validation of the library. This has been in two areas, the first of which is the correct operation of the library. The second is perhaps not as obvious but is just as important, the usability of the library. If the symbol table library is difficult to use or incorporate within a compiler then it is of little use. The objectives therefore were to uncover any errors in the library and prove that the library is simple and easy to use. In the previous section the testing phases of the implementation were outlined, the following is an in-depth discussion.

## 6.1 Testbed.

This is the most simple of the testing routines, it is primarily concerned with the debugging process. Command files can be constructed that mimic any sequence of symbol table manipulations, all results being dumped to output files for analysis. All routines were initially debugged and tested using this testbed. The main reason for using this testing method was because it allowed the isolation and testing of features separately. Interactions with other features could slowly be phased in as a routine's soundness was proven.

This isolationalist approach to testing is valid in this case, because of the nature of a symbol table. Symbol table accesses are essentially independent, and so there is not the problem that some special sequence of commands will put the table into an incorrect state. Once the underlying table structure is sound it only needs to be shown that each individual routine operates correctly and does not corrupt the structure. After debugging with this testbed it was necessary to begin testing within a real test environment.

## 6.2 Example Pascal parser.

It is perhaps fitting that automatic generation tools were used to test the library, it was decided not to use the ACK tools because of previous experience with the standard Unix tools Yacc and Lex. Wirth's original BNF definition for Pascal was used to design my Yacc input.

The grammar had to be altered slightly to make it acceptable as Yacc input. This was because the grammar had been expanded to make it easier to read, thereby introducing some ambiguities. A typical example of this is in expressions, where a terminal symbol had been expanded from 'identifier' to 'constant identifier', 'variable identifier' etc. All of these expanded productions went to the single terminal 'identifier'. These grammatical conflicts are unresolvable within the Yacc generated parser, the solution was however relatively simple. The alterations made to the grammar did not change the language.

It was decided however to make some deliberate alterations to the grammar, these were for pragmatic reasons. Some forms of type declarations were excluded because they tested nothing unique and would have required an unworthwhile amount of time to implement. The features of Pascal that were left out were files, real numbers, variant records and complex types composed of other complex types. It is believed that these omissions do not detract from the validity of the tests.

Initially the full type definitions were built up as a program was parsed, so that full type checking could be done for every variable, function etc. access. However this quickly proved to be too large an undertaking. More importantly this function of a compiler has no connection to the use of the symbol table. Certainly the symbol table is looked up to check for existance of a referenced type declaration but this is merely an existence check. The actual complex type does not have to be built. It was decided to restrict checking to existential checks within declarations and expressions. Some examples;

-making certain that constants, types and variables are declared before use,

-checking that types exist for all variable declarations,

-checking that referenced fields within records exist,

-loop counters in 'for' loops declared local.

As each identifier is encountered within the parser it is inserted locally, i.e. it is inserted if and only if it does not already exist in the current scope. This is a single call to the symbol table. Similarly lookup is a single call, as are the routines

required to implement the Pascal 'with' statement and referencing a record field.

One symbol table option is to preload the symbol table with a module containing language types, constants etc. Although strictly not necessary it was decided to use this feature. The symbol table is therefore loaded with Pascal predefined identifiers e.g. integer,char at the start of the program parse.

The results were convincing. There were no problems incorporating the symbol table routines with the Yacc generated parser and Lex generated scanner. Most accesses to the symbol table required only one function call, only using a fraction of the available options. Confidence was gained that at the very least the symbol table library is more than adequate for languages such as Pascal. The library was also proved to be simple and easy to use in practice. A listing of the Pascal parser and scanner is included in Appendix D.

## 6.3 Modula Parser.

Following the success with the Pascal parser it was decided to try some of the advanced features, these revolve mainly around modules. Although modules were shown to be functioning in the Pascal example (by the use of a module to preload the symbol table) they deserved a more rigorous testing. It was intended to use Modula2 for this example because of its complex module facilities and compact syntax. Because of it's similarity to Pascal it was decided to add the module structures of the Modula2 BNF grammar to the already constructed Pascal parser. Therefore although not strictly Pascal or Modula2, the resulting grammar allows the testing of a number of Modula2 features.

The conversion to Modules was relatively straight forward, the major problem was interpreting the Modula2 standard. Modula2 is a developing and changing language, consequently it is difficult to find non conflicting references. Wirth is particularly obtuse about some features, with statements such as,

> "Objects local to a module are said to be at the same scope level as the module. They can be considered as being local to the procedure enclosing the module but residing within a more restricted scope"

The Modula parser is included in the separate folder which contains all code and test listings. The module features that were implemented included;

-Imports and exports through multiple levels of nested modules,

-DEFINITION and IMPLEMENTATION modules with IMPORT and EXPORT lists,

-Restricted, closed scope lookups within modules,

-Selective or total import from separately parsed modules,

-Saving and reloading module symbol tables.

It is evident that the Modula parser uses a greater number of the symbol table routines supplied by the library than the Pascal parser. Some of the major features used are listed below. Note that in all these cases the method of implementing the language feature was not necessarily the most efficient, or only way of, achieving the desired result.

Symbol iteration was used to process exports when closing nested modules.

Reseting of symbol wildcards was used when exports from nested modules changed from being exported to 'normal' symbols. This occurs when an export reaches the outermost level of it's exportation. For example,

```
IMPLEMENTATION MODULE nest;
    MODULE nest1;
        MODULE nest2;
        EXPORT birdy;
            MODULE nest3;
            EXPORT birdy;
            CONST
                birdy = 10;
            BEGIN
            END nest3;
        BEGIN
        END nest2;
    BEGIN
    (*  'birdy' is defined here but it must not be exported
        out a further level, so the symbol must be changed
        from EXPORT. *)
    END nest1;
BEGIN
END nest.
```

Multiple lookup and symbol deletion were used to delete the forward references of exports from nested modules. For example, in the above example there are two symbols called `'birdy'` in the body of module `nest2`, one being the forward declaration, the second the actual symbol that has been imported from module `nest3`. This occurrence is handled by finding and deleting the forward reference.

Restricted lookup was used for all searching within modules, each module was treated as a closed scope during the lookup process.

The various import routines were used to implement imports, both selective

```
FROM otherworldModule IMPORT anAlien;
```

and total

```
IMPORT otherworldModule;
```

All module input and output were accomplished using the symbol table input/ output routines.

These are just some examples from the Modula parser. There are a large number of semantic checks that must be incorporated in languages that use modules. In the Modula2 demonstration parser used for testing the symbol table library a number of semantic checks have not been implemented. Only those checks that were considered necessary to adequately test the library have been implemented.

Appendix C contains a simple test that was undertaken using the Modula parser. The three modules were in separate files, the two definition modules were parsed first so that their modules would be available for import into the program module.

# Section Seven.

## 7.0 Efficiency experiments.

The symbol table library must be efficient in its general operation to be useful to the compiler writer. A user will be cautious about using a module which is considered inefficient, and may effect the overall efficiency and execution time of the compiler. To test this area it was decided to concentrate on the two aspects which were thought to most effect the speed of the symbol table operation. These are the average execution times of the two principle routines, lookup and insert, and the effectiveness of the hashing function.

These were selected for examination and tuning because every routine uses one of these two functions on accessing the symbol table, and a hash table's efficiency depends mainly on the length of the collision chains, hence it's hashing function.

### 7.1 Measuring Execution Speed.

The Unix utility gprof along with the C compiler option -pg was used to generate an execution profile for a simple test program. The test program used the symbol table library to scan text files for strings, inserting them into a table. Various statistics which are collected at the same time, were used in analysis of the hash function.

Gprof displays the call graph profile built during program execution. It includes

-number of times each routine is called

-average time for each call

-average time spent in each of the routine's children (i.e. those functions that are called by the routine).

Analysis of this information allows a detailed picture to be built of the area where the most execution time is used. Simple strategies can then be implemented to try and decrease these.

A number of tests were devised, initially a number of runs were executed with the first test to try and ascertain how accurate the gprof results were. The results over multiple runs were found to be very similar, therefore the gprof results from a

series of tests could realistically be used in a comparative study.

The general method was to execute lookup and insert many times, and examine the results to determine where time could be saved. Then a comparison would be made with equivalent routines in a production compiler to see how the developed routines fare.

The results of the experiments are displayed as a series of tables. Each table lists the subfunctions called by the routine under examination. The percentage of time spent executing these subfunctions compared with the total time spent executing the routine is detailed. The percentage time spent within the actual routine is listed under 'self'.

In each table the 'before' column gives the percentage of time spent in each subfunction before any changes were made, the percentages after any optimisations are displayed in the 'after' column.

Double quotes in the subfunction column indicates a routine that has been merged with the subfunction listed above it.

### 7.1.1 Insertion.

The insertion process always takes about the same amount of time, as the same sequence of instructions are always executed. The time is only altered by the string copying process, so it will take slightly different times depending on the length of the identifier being hashed. Therefore insertion is a simple case to examine; there is no best or worst case involved.

For the tests on the insertion process identifiers were generated by scanning through the Unix csh manual entry. The strings gathered from this were not unique, but this was not a concern for insertion. Approximately 13000 identifiers were inserted into the symbol table during the test runs.

A number of interesting observations were made. The results, along with the results of the same experiment after optimising, are given in Table 7.1. It was found that most of the time taken was in allocating storage space for the identifier that was being copied: this, combined with actually copying the identifier, was taking half the time in each call. This was as a result of calling the storage allocator malloc(), each time space was required for a string.

**Table 7.1 Testing the Insertion function.**

| Before | | After | |
|---|---|---|---|
| Subfunction | %time | Subfunction | %time |
| Malloc() | 42 | mystralloc() | 20 |
| strcpy() | 6 | " | |
| strlen() | 2 | strlen() | 12 |
| calc_hash() | 24 | calc_hash() | 35 |
| link_next() | 5 | link_next() | 8 |
| self | 17 | self | 24 |

**Average time for insert**

**Before: 0.15 ms (0.12 ms in subfunctions)**
**After: 0.11 ms (0.9 ms in subfunctions)**

To overcome this a routine was built on top of the standard malloc(). The routine keeps a large buffer of memory space which is allocated for identifiers as the need arises, the actual memory allocation routine is only called when the buffer is exhausted. The routine also copies the identifier, saving the overhead of an additional procedure call.

The resulting saving was worthwhile, an approximately 25% decrease. At this point it was obvious that the calculation of the hash value needed attention as well, the details of this are in 7.1.3.

### 7.1.2 Lookup.

Lookup is the most important routine as it will typically occur more often than insertion. The lookup time varies depending on the length of the hash table collision chains. The best and worst case must therefore be examined to give a true indication of the efficiency of the routine.

Best case analysis is simple, all that is required is to be certain that the identifier being searched for is at the start of the collision chain. This is guaranteed if it was the last identifier inserted. Worst case is more complex as it relies on the number of identifiers in the table, and the hash function. Therefore a general worst case figure cannot be calculated, only a worst case for the test file and hash function that is being used. In the following tests a dictionary of unique identifiers (approximately 24000) is used and the worst case is simulated by looking up each identifier before it is inserted. As the identifiers are unique, the

entire collision chain must be searched at each lookup.

The large number of identifiers that are being hashed to the table are rather unrealistic. The hash table becomes seriously overloaded and performance deteriorates, as the worst case results in Table 7.3 show. Without overloading it would be expected that the behaviour would be close to the best case analysis, however this depends to a large extent on the efficiency of the hash function.

**Table 7.2 Best case Lookup.**

| Before | | After | |
|---|---|---|---|
| Subfunction | %time | Subfunction | %time |
| self | 30 | self | 47 |
| strcmp() | 18 | " | |
| calc_hash() | 52 | calc_hash() | 53 |

**Average time for lookup**

**Before: 0.11 ms**

**After: 0.10 ms**

**Table 7.2 Worst case Lookup.**

| Before | | After | |
|---|---|---|---|
| Subfunction | %time | Subfunction | %time |
| self | 41 | self | 91 |
| strcmp() | 54 | " | |
| calc_hash() | 7 | calc_hash() | 9 |

**Average time for lookup**

**Before: 0.93 ms**

**After: 0.53 ms**

The results of the test are shown in Tables 7.2 and 7.3. These tables show the average timings before and after various optimisations were carried out. They show that calculating the hash value required the most time in the best case, string comparisons the most in the worst case. The results in the 'after' column show the decrease in execution time by including string comparison inline, using register variables. This small change dropped the worst case lookup to nearly half

its original value. As would be expected, this change had minimal effect on the best case.

### 7.1.3 Hash value calculation.

In both lookup and insertion calculating the hash value was now taking an appreciable amount of time. The simple addition of register variables made an enormous improvement as shown in Table 7.4. Table 7.5 shows the overall percentage decreases in the two main routines, from the simple tuning carried out. The decrease illustrates just how susceptible programs are to careful tuning.

**Table 7.4 Hash value calculations.**

| Before | | After |
|---|---|---|
| Function | Av time (ms) | Av time (ms) |
| calc_hash() | 0.05 | 0.02 |
| insert()<br>lookup() -Best<br>lookup() -Worst | 0.11<br>0.10<br>0.54 | 0.08<br>0.08<br>0.50 |

**Table 7.5 Overall Insert and Lookup improvement**

| Before | | After | |
|---|---|---|---|
| Function | Av time (ms) | Av time (ms) | Percent decrease |
| insert()<br>lookup() -B<br>lookup() -W | 0.15<br>0.11<br>0.93 | 0.08<br>0.08<br>0.50 | 47%<br>27%<br>46% |

### 7.1.4 C input routines.

Fscanf is the standard C file oriented input routine. It was suggested that perhaps it would not be efficient, and could be improved upon by a purpose written routine. The decision was therefore made to write a routine and test it against the standard scanf routine.

The results, as depicted in Table 7.6, were fairly conclusive. The routines, fscanf and my own, both gave the same average execution time. It must be noted however that the purpose written routine will allow the input of any length of string and therefore calls the storage allocator within the input function. The fscanf function does not have to allocate storage, as one of it's parameters is a pointer to already allocated storage. This allocation of storage takes approximately a third of the time. The average time without allocating storage is also given in Table 7.6.

**Table 7.6 C Input comparison.**

| Function | Av time (ms) |
|---|---|
| fscanf() | 0.13 |
| my_scan() | 0.13 |
| my_scan() -Without malloc() | 0.08 |

In view of the many options provided by the scanf routine, and the fact that it is not actually used very much within the symbol table, it was decided to retain its use.

### 7.1.5 Comparison with the ACK C compiler.

The savings in execution time have been detailed so far, but these figures are not very helpful without an equivalent set of routines from an actual compiler, for comparison.

The ACK C compiler, was selected to find the typical execution times for the lookup and insertion routines, for a number of reasons:

-simple to copy and recompile the compiler with the profiling option on

-uses the same hashed symbol table structure

-widely distributed production compiler

After rebuilding the C compiler with the profiling flag set, several of the symbol table library files were recompiled and the gprof output examined.

Ideally, analogous routines to the library insert and lookup routines were required for comparison. The ACK C compiler did not contain these. The two routines found (by examining the code), that were closest to the insertion and lookup routines were 'declare_idf()' and 'str2idf()'. Table 7.7 gives an indication of their average run times, compared with the symbol table routines.

**Table 7.7 Comparison with ACK C compiler**

| Function | Av time (ms) |
|---|---|
| str2idf()<br>lookup() | 0.01<br>0.08 |
| declare_idf()<br>insert() | 0.13<br>0.08 |

As can be seen, the results are not very comparable.

In the insertion case, the ACK routine achieves the same as the library routine and additionally stores information about the symbol, for this reason it is slower.

In the lookup case, the ACK routine merely iterates along the collision chain, no limitations on the lookup are provided at all. The ACK routine does not calculate the hash value of the symbol, this passed in as a parameter. As has been demonstrated this takes a considerable amount of time in the library lookup routine.

The most that can be gained from this comparison is that the library routines are at the very least, of the same order of magnitude in execution speed, as the routines in an actual compiler. Comprehensive testing was not undertaken because of the differences in the routines involved, however this does not detract from these results. Confidence has been gained that the library routines execute at a useful speed.

## 7.2 Finding an efficient hash function.

As previously mentioned, the hash function is an important consideration in the efficiency of any hash based structure.

### 7.2.1 An ideal hash function.

An ideal hash function is an elusive beast. There are a number of attributes we require from a hash function,

-The hash value must be the same all the time. This may sound trivial, but this means that the values must be the same from one program

invocation to the next, requiring that masks and random numbers not be generated from system clock times, for example.

-To be useful the function must be quick to calculate

-The hash function must spread the identifiers evenly over the hash range.

### 7.2.2 Initial Hash Function.

While developing the symbol table library the standard Ack hash function was used. All that was required during the development phase was a function that worked, it's efficiency was not an issue. The hash function in the ACK library basically does the following,

-initially generates a random mask with a congruence generator for use with all identifiers.

-to calculate a hash value, the characters of the identifier are x-ored with the mask and summed.

-the resulting number is divided modulo TABLESIZE to give a number in the required range.

### 7.2.3 Testing the hash function.

At a later stage in the project the hash function was put through a series of tests. This was done as part of the general efficiency testing of the module. It was expected that the hash function would require little attention, having been taken from the ACK library.

The results for each test carried out on the hash function are displayed using two graphs. The histogram graphs display the length of the hash chains (in steps of 10), verses the number of hash buckets that have chain lengths in that range.

The line graphs display the symbol table's hash bucket range (in steps of 25), verses the number of identifiers hashed to each subrange.

The accuracy of these graphs is not very high as subranges have been used rather than each individual data value, however they adequately display the trends in the data gathered from the tests.

Initially tested with the table size of 256 buckets, the results of hashing 24000 unique identifiers are shown in Graphs 7.1 and 7.2. Graph 7.1 shows that the number of hash buckets with hash chain lengths greater than 35 is almost constant. The last interval in the graph is for hash chain lengths greater than 190 identifiers, the longest chain having 218 identifiers. It would be expected that the

distribution would decrease for the longer hash chains.

The distribution is obviously not random, the polarity around 0 and 255 is very noticeable in Graph 7.2. This led to the testing of a larger table size to see if the polarity would accentuate.

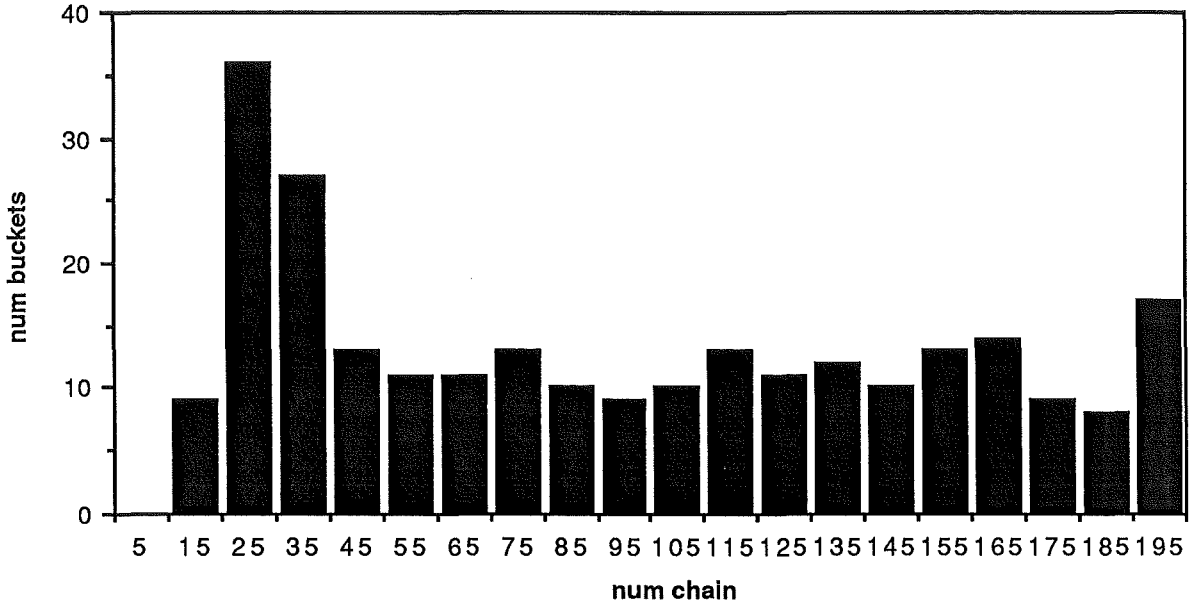
The results with the same 24000 identifiers and a table size of 1024 are in Graphs 7.3 and 7.4. Clearly the trends shown in Graphs 7.1 and 7.2 have been followed. The polarity in Graph 7.4 is extreme, 700 of the 1024 buckets have no identifiers hashed to them. Graph 7.3 shows the same constant distribution of the longer length hash chains as was demonstrated in Graph 7.1.

There is obviously something wrong with the function, it certainly does not evenly spread the identifiers over the full range. A new hash function was obviously needed but it was decided not to discard the ACK function entirely, but rather experiment with it.

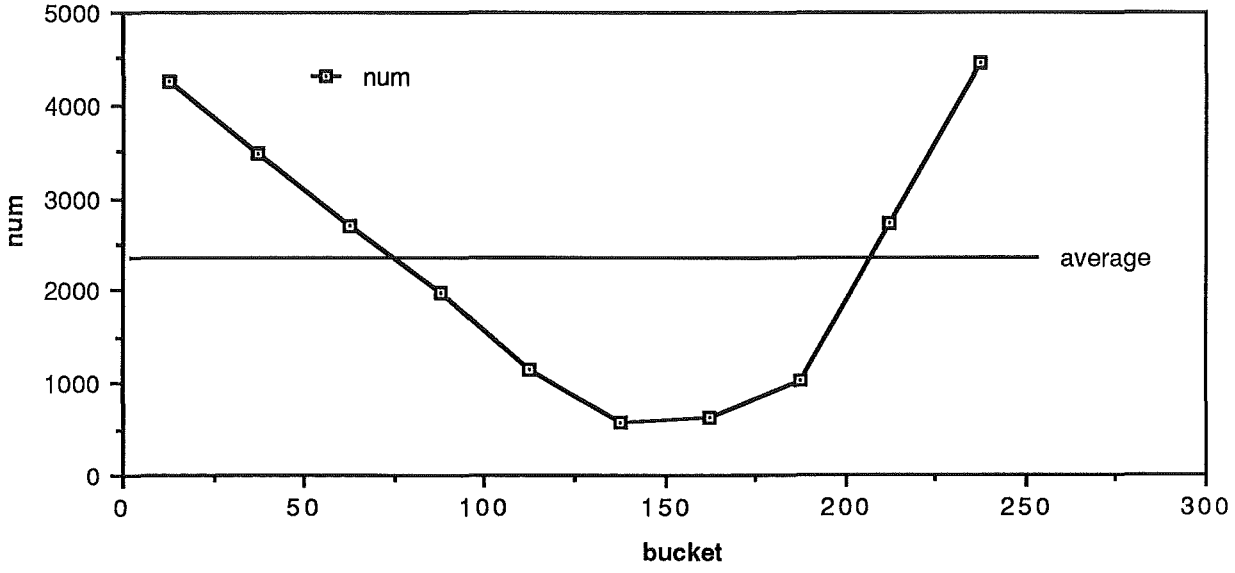
Firstly experimenting with the mask was undertaken, verifying that random numbers were generated. When this was proved, it was decided to try replacing the random mask with a simple sequence i.e. 1,2,3,..., one digit for each location in the mask. As the results of hashing with this mask were better, this approach was continued. After a number of other random experiments the best results were as given in Graphs 7.5 to 7.8. The mask used was a sequence of two alternating prime numbers. The method of coming up with this mask is definitely not desirable, but as mentioned in Section 7.2.4 this will hopefully improve.

Graph 7.5 shows a normal distribution about an average of 94.5 for the hash chain lengths. Graph 7.7 also displays a more desirable distribution for the hash chain lengths, the number of buckets with long hash chains decreases rapidly. Graphs 7.6 and 7.8 also show better distributions, being centered closer around the expected mean. However the distribution in Graph 7.8 still displays an undesirable trend, i.e. the low number of identifiers hashed to the lower third of the table.

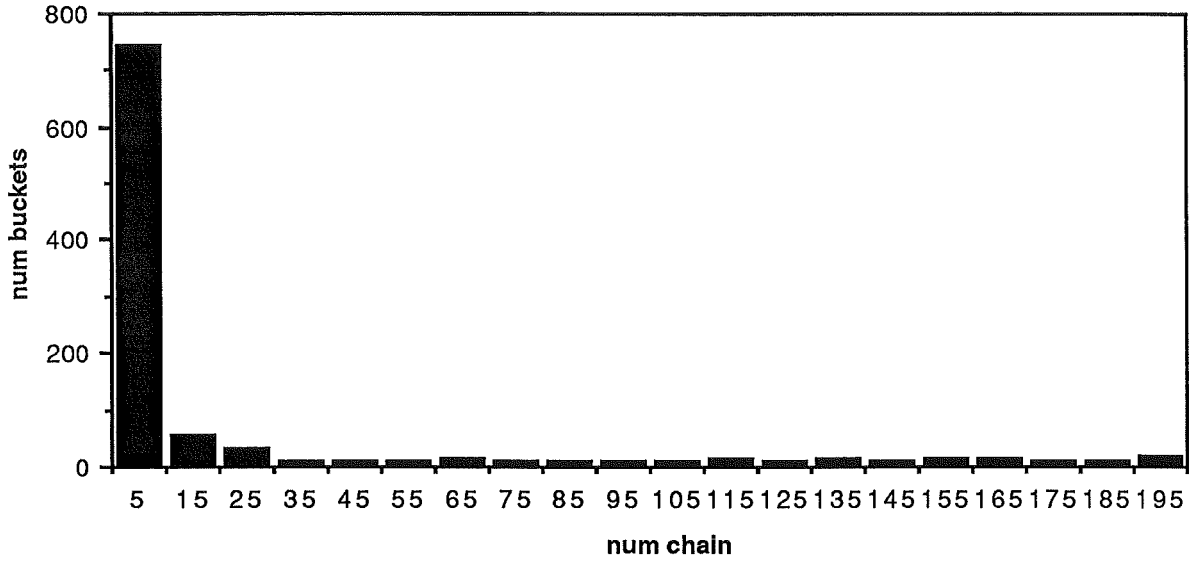
**Graph 7.1 Number of hash buckets verses the length of the hash chains for hash table with 256 buckets. Original ACK hash mask.**



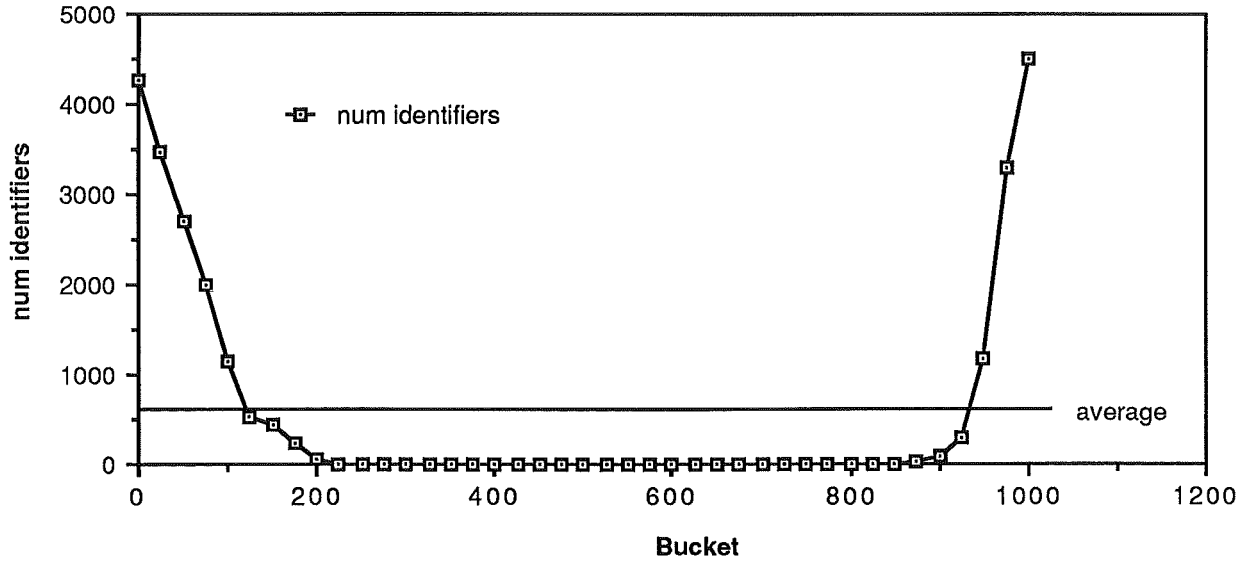

**Graph 7.2 Number of identifiers hashing to each 25 bucket interval in a 256 bucket range. Original ACK hash mask.**



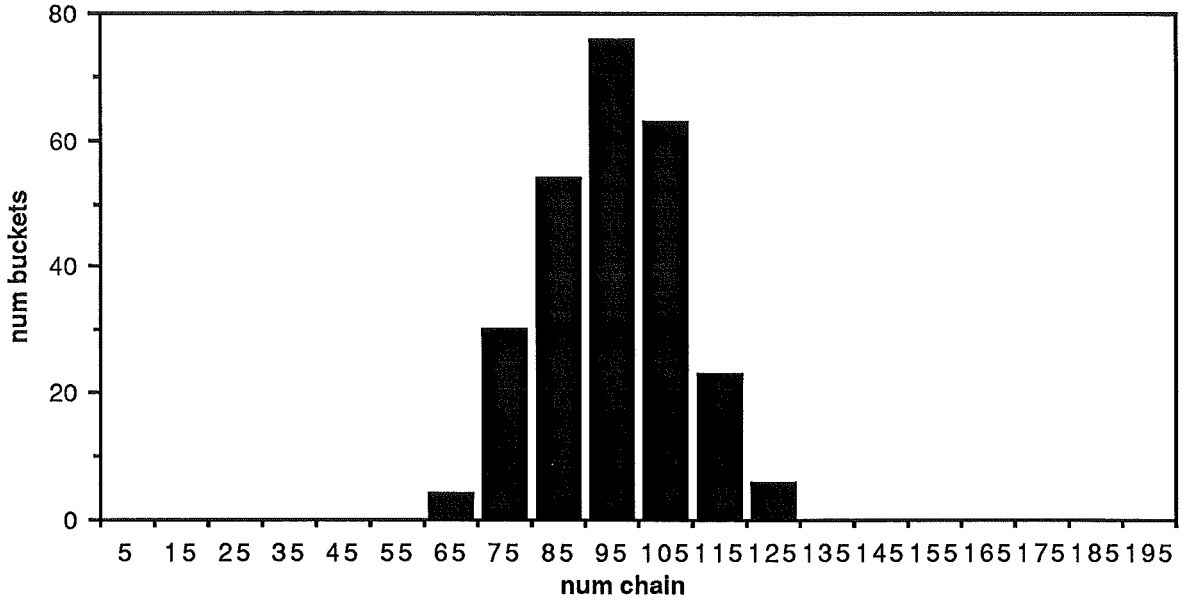
**Graph 7.3 Number of hash buckets verses the length of the hash chains for hash table with 1024 buckets. Original ACK hash mask.**



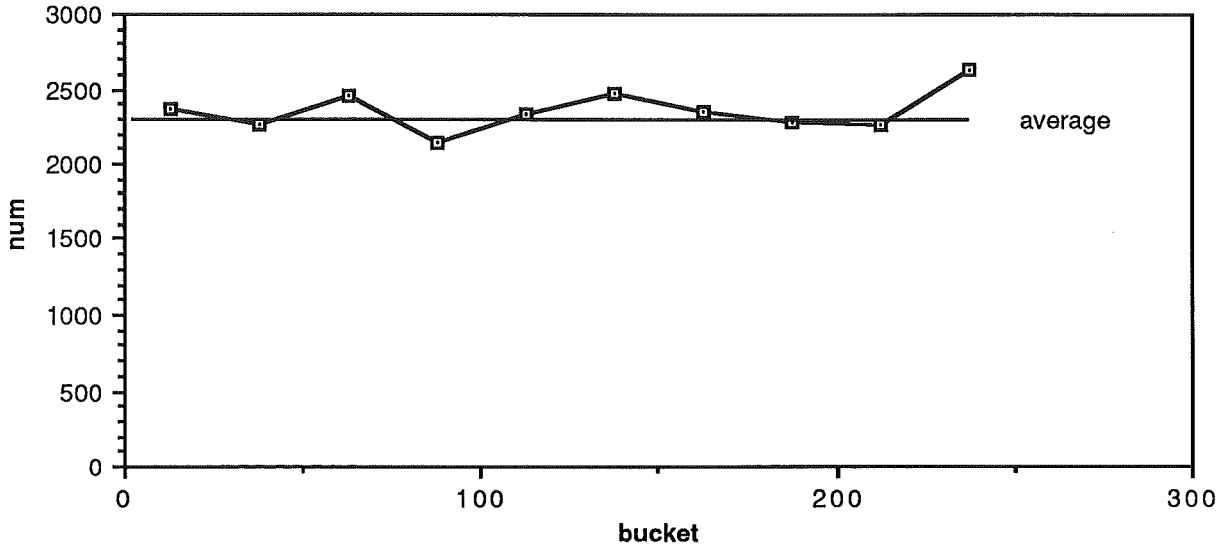

**Graph 7.4 Number of identifiers hashing to each 25 bucket interval in a1024 bucket range. Original ACK hash mask.**



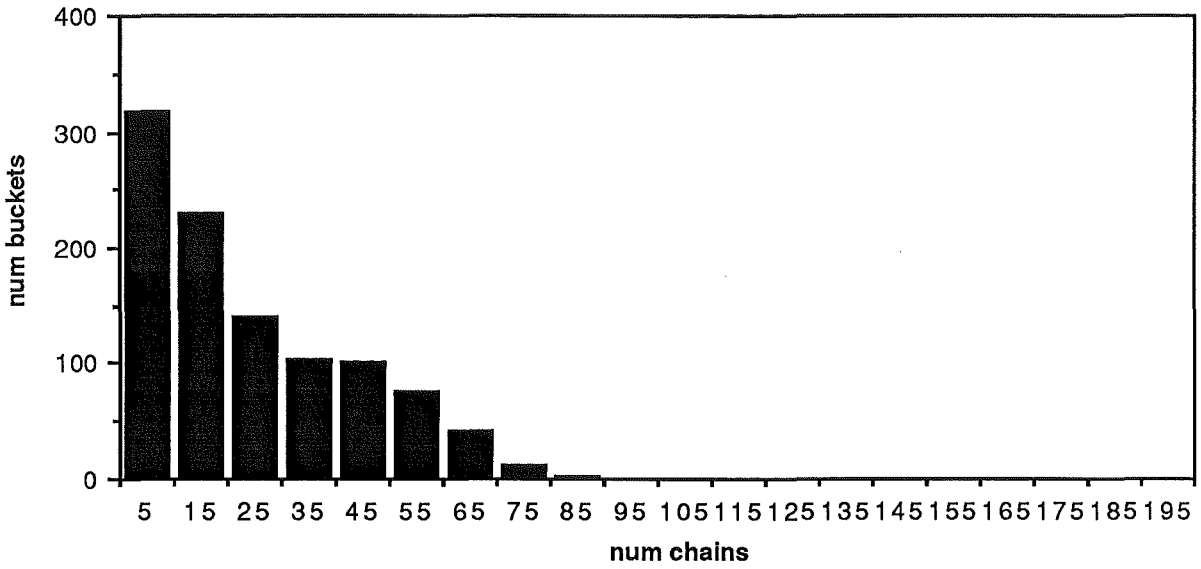
**Graph 7.5 Number of hash buckets verses the length of the hash chains for hash table with 256 buckets. Modified hash mask.**




**Graph 7.6 Number of identifiers hashing to each 25 bucket interval in a 256 bucket range. Modified hash mask.**



**Graph 7.7 Number of hash buckets verses the length of the hash chains for hash table with 1024 buckets. Modified hash mask.**




**Graph 7.4 Number of identifiers hashing to each 25 bucket interval in a 1024 bucket range. Modified hash mask.**

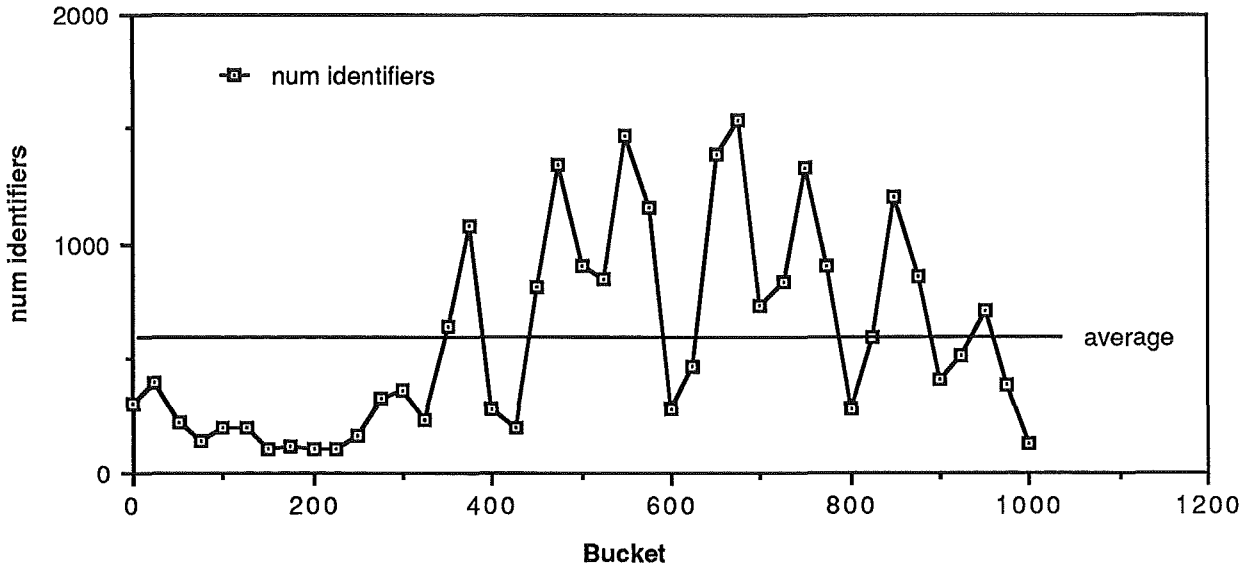


Table 7.8 gives a brief statistical breakdown of the characteristics between the standard ACK mask and the mask which was eventually used.

**Table 7.8 Hash mask statistics.**

| Mask | Table size | Max chain | Min chain | Average | Std Deviation |
|---|---|---|---|---|---|
| Standard | 256 | 218 | 15 | 94.5 | 58.5 |
| Standard | 1024 | 218 | 0 | 23.6 | 50.0 |
| Modified | 256 | 126 | 67 | 94.5 | 12.4 |
| Modified | 1024 | 84 | 0 | 23.6 | 18.7 |

### 7.2.4 Further analysis of hashing problem.

The uneven distribution of hash values as illustrated in Figures 7.1 and 7.2 is the subject of further research within this department.

The cause of the problem is now understood, and investigation into the properties of other hash functions may be undertaken, the aim being to test and recommend the better of these hash functions.

It is envisaged that when this occurs, the best of the tested hash functions will be incorporated into the symbol table library. This would require the replacement of two simple routines.

# Section Eight.

## Summary.

The aim of this project was to develop a general purpose symbol table library for use within the compiler development framework of the Amsterdam Compiler Kit. This has been successfully achieved and the resulting library has been extensively tested and tuned to provide a useful addition to the Amsterdam Compiler Kit.

Uses of the library are not restricted to the Amsterdam Compiler Kit, as was clearly demonstrated during the testing phases. The library was successfully used with the Unix compiler writing tools Yacc and Lex, and on a more general level could be used wherever an identifier based retrieval structure is required.

# References.


[TAN80] A.S.Tanenbaum, H.van Straveren, E.G.Keizer, and J.W.Stevenson,
'A Practical Tool Kit for Making Portable Compilers'
Mathematics Dept. Vrije Universiteit Amsterdam, The Netherlands.

E.H.Baalbergen, D.Grune, and M.Waage,
'The CEM Compiler'
Mathematics Dept. Vrije Universiteit Amsterdam, The Netherlands.

C.Jacobs,
Source code of the ACK Modula2 compiler
Amsterdam Compiler Kit.

B.W.Kernighan and D.M.Ritchie,
*The C Programming Language*
Prentice-Hall, 1978.

S.C.Johnson,
'Yacc - Yet Another Compiler-Compiler'
Bell Laboratories, July 1975.

M.E.Lesk,
'Lex - A Lexical Analyser Generator'
Bell Laboratories, October 1975.

[WIR83] N.Wirth,
*Programming in Modula2*
Springer-Verlag, 1983.

A.V.Aho and J.D.Ullman,
*Principles of Compiler Design*
Addison Wesley, 1977.

D.E.Knuth,
*The Art of Computer Programming Vol. 2*
Addison Wesley, 1973.

A.T.Schreiner and H.G.Friedman Jr,
*Introduction to Compiler Construction with Unix.*
Prentice-Hall, 1985.

K.Jensen and N.Wirth,
*Pascal User manual and Report*
Spinger-Verlag, 1977.

Department of Defence,
*Reference Manual for the Ada Programming Language*
United States Department of Defense, March 1983.

[GRA79] S.L.Graham, W.N.Joy and O.Roubine,
'Hashed symbol tables for languages with explicit scope control'
Sigplan Notices vol. 14, no. 8, pp 50-7.

[REI83] S.P.Reiss,
'Generation of Compiler Symbol Processing Mechanisms from Specifications'
T.O.P.L.A.S vol. 5, no. 2, pp. 127-63.

[POP78] G.J.Popek, J.J.Horning, B.W.Lampson, J.G.Mitchell, and R.L.London,
'Notes on the design of Euclid'
Sigplan Notices vol. 13 pp. 11-17.

R.J.LeBlanc and C.N.Fisher,
'On Implementing Separate Compilation In Block-Structured Languages',
Sigplan Notices vol 14 pp. 139-43.

M.I.Blower,
'An Efficient Implementation of Visibility in Ada'
Sigplan Notices Vol. 19, No. 6, pp. 259-65.

[HAN81] D.R.Hanson,
'Is Block Structure Necessary?'
Software Practice and Experience Vol. 11 1981 pp. 853-66.

[CLA80] L.A.Clarke, J.C.Wileden, and A.L.Wolf,
'Nesting in Ada Programs is for the Birds'
Sigplan Notices Vol. 15 no. 11 pp. 139-45.

[ROD88] Rod Harries,
*Das Anna Magdalena Bach Klavier Buch*
Unpublished.

## NAME

**SCOPE MANIPULATION**
new_symbol_table, init_hash, open_scope, cur_open_scope, cur_open_anon_scope, close_scope, cur_close_scope, relink_scope_of, read_scope, save_scope, dispose_scope, dispose_saved_of, restore_scope_of, restore_scope, widen_scope, current_scope_name_of, current_scope_name

**LOOKUP**
look, lookup, lookup_local, lookup_record, lookup_in_scope_of, lookup_in_scope, search_saved, multiple_lookup_in, multiple_lookup, next_lookup

**SYMBOL MANIPULATION**
insert, insert_wild, insert_local, iterate_start, sym_iterate, reset_wild_of, reset_wild, delete_sym

**MODULE MANIPULATION**
import_preload_to, import_preload, import_module_to, import_module, import_from_module_to, import_from_module, import_generics_to, import_from_generics, import_exports, import_from_exports

**DEBUGGING**
dump_table, snap_shot

## SYNOPSIS

**SCOPE MANIPULATION**

```
struct sym_table    *new_symbol_table()

init_hash()

struct scope_node   *open_scope(sc_type,sym_table,scope_id)
                    FLAG sc_type;
                    struct   sym_table *p_table;
                    char *scope_id;
struct scope_node   *cur_open_scope(scope_id)
                    char *scope_id;
struct scope_node   *cur_open_anon_scope()

struct scope_node   *close_scope(sym_table)
                    struct   sym_table *p_table;
struct scope_node   *cur_close_scope()

struct scope_node   *relink_scope_of(scope_node,sym_table)
                    struct scope_node   *p;
                    struct sym_table    *p_table;
struct scope_node   *widen_scope(scope_node)
                    struct scope_node   *p;

struct scope_node   *read_scope(mod_name)
                    char *mod_name;

save_scope(p)
                    struct   scope_node    *p;
```

```
dispose_scope(p)
        struct scope_node  *p;

dispose_saved_of(name,p_table)
        char *name;
        struct sym_table  *p_table;

struct scope_node  *restore_scope_of(scope_name,p_table)
        char *scope_name;
        struct sym_table  *p_table;
struct scope_node  *restore_scope(scope_name)
        char *scope_name;

char *current_scope_name_of(p_table)
        struct sym_table  *p_table;
char *current_scope_name()
```

LOOKUP

```
SYMNODE_TYPE  *look(id,p_table,starts,stops,multiple)
        char *id;
        struct sym_table  *p_table;
        struct scope_node  *starts;
        struct scope_node  *stops;
        int multiple;
SYMNODE_TYPE  *lookup(id)
        char *id;
SYMNODE_TYPE  *lookup_local(id)
        char *id;

SYMNODE_TYPE *lookup_record(p,id_name)
        struct scope_node  *p;
        char *id_name;

SYMNODE_TYPE  *lookup_in_scope_of(id_name,sc_name,p_table)
        char *id_name, *sc_name;
        struct sym_table  *p_table;
SYMNODE_TYPE  *lookup_in_scope(id_name,scope_name)
        char *id_name, *scope_name;

struct scope_node  *search_saved(scope_name,p_table)
        char *scope_name;
        struct sym_table  *p_table;

SYMNODE_TYPE  *multiple_lookup_in(start,name,p_table)
        char *start, *name;
        struct sym_table  *p_table;
SYMNODE_TYPE  *multiple_lookup(start,name)
        char *start, *name;
SYMNODE_TYPE  *next_lookup(name)
        char *name;
```

SYMBOL MANIPULATION

```
SYMNODE_TYPE  *insert_wild(id,p_table,wild)
        char *id;
```

```
                struct sym_table   *p_table;
                int wild;
SYMNODE_TYPE  *insert(id)
                char *id;

SYMNODE_TYPE  *insert_local(id)
                char *id;
iterate_start(p,wild)
                struct scope_node  *p;
                int wild;

SYMNODE_TYPE  *sym_iterate(p)
                struct sym_node    *p;

int reset_wild_of(id,new_wild,old_wild,p_table)
                char *id;
                int new_wild, old_wild;
                struct sym_table   *p_table;
int reset_wild(id,new_wild)
                char *id;
                int new_wild;

int delete_sym(id,u_node,p_table)
                char *id;
                SYMNODE_TYPE  *u_node;
                struct sym_table   *p_table;
```

MODULE MANIPULATION

```
int import_preload_to(pre_name,p_table)
                char *pre_name;
                struct sym_table   *p_table;
int import_preload(pre_name)
                char *pre_name;

struct scope_node  *import_module_to(m_name,p_table,gen_type)
                char *m_name;
                struct sym_table   *p_table;
                int gen_type;
struct scope_node  *import_module(mod_name);
                char *mod_name;

struct scope_node  *import_generics_to(p,p_table,gen_type)
                struct scope_node  *p;
                struct sym_table   *p_table;
                int gen_type;
struct scope_node  *import_exports(p)
                struct scope_node  *p;

SYMNODE_TYPE  *import_from_module_to(mod_n,id_n,p_table,gen_type)
                char *mod_n, *id_n;
                struct sym_table   *p_table;
                int gen_type;
SYMNODE_TYPE  *import_from_module(mod_name,id_name)
                char *mod_name, *id_name;
```

SYMNODE_TYPE ***import_from_generics_to**(p,id_n,p_table,gen_type)
struct scope_node *p;
char *id_n;
struct sym_table *p_table;
int gen_type;
SYMNODE_TYPE ***import_from_exports**(p,id_name)
struct scope_node *p;
char *id_name;

**DEBUGGING**
**dump_table**(d_nodes,p_table,filename)
FLAG d_nodes;
struct sym_table *p_table;
char *filename;

**snap_shot**()

## DESCRIPTION

These routines provide a general purpose symbol table manipulation package. When compiled, the package provides the user with an object file which allows the insertion and manipulation of symbol data. The user is oblivious to the underlying structure of the table, data is maintained within the symbol table by the provided routines.

This library is designed to be used within the Amsterdam Compiler Kit, (see ack), maintaining and manipulating a hash structured symbol table. The symbol table characteristics and various optional parameters are set by the user. These determine the structure of the underlying symbol table.

One of the characteristics that must be specified is the user defined struct that will be used to store the user information within the symbol table. This structure is defined as SYMNODE_TYPE, a number of routines, noticeably lookup and insertion, return pointers to instances of these structs.

The routines listed in the synopsis include a number of aliased functions. The description of these routines is included in the description of the function which they alias. These aliased routines take default values for a number of parameters which it is forseen will not often be needed. An example of this is the specification of the symbol table to work on. Many applications will need only a single symbol table, so the aliased routines use the user specified default, DEFAULT_TABLE.

### SCOPE MANIPULATION

*New_symbol_table* returns a pointer to a newly allocated symbol table struct.

*Init_hash* initialises the hash masks. This must be done before any calls to the symbol table routines.

*Open_scope* opens a new scope in the indicated symbol table with the given name. The scope can be either open or closed (indicated by using the predefined constants OPEN_SCOPE or CLOSED_SCOPE). A closed scope disallows references to external scopes by restricting the lookup process. Program blocks with explicit interfaces, such as modules/ packages, may be specified in this way. A pointer to the new scope is returned.

Cur_open_scope opens an OPEN_SCOPE with the specified name in the

DEFAULT_TABLE.

*Cur_open_anon_scope* opens an anonymous OPEN_SCOPE in the DEFAULT_TABLE.

*Close_scope* closes the current scope in the indicated table. If the scopes are being saved, (a user option), the scope is saved and a pointer to it is returned, else a NULL pointer is returned.

*Cur_close_scope* closes the current scope in the DEFAULT_TABLE.

*Relink_scope_of* relinks the given scope into the indicated symbol table as the new current scope. Returns a pointer to the scope linked into the table, NULL on error.

*Widen_scope* makes the indicated scope the current scope of the DEFAULT_TABLE.

*Read_scope* reads in the previously saved scope from secondary storage. Returns a pointer to the reconstructed scope, NULL on error. The module manipulation routines are implemented using this routine and *save_scope.*

Both *read_scope* and *save_scope* require the user to provide routines to read or save the user data stored within the user defined struct. The format for these routines is outlined in *user_code.c.*

*Save_scope* saves the indicated scope to secondary storage. The file naming structure indicated in *user_def.h* is used. True or false is returned to indicate success or failure. Failure would indicate that a file error occurred.

*Dispose_scope* disposes of the indicated scope. This must not be currently part of the symbol table, nor a saved scope.

*Dispose_saved_of* allows the user to dispose of a saved scope by referring to it by name. If the scope exists in the saved scopes the scope is delinked from the structure and the scope is deallocated The allocation and deallocation is handled by the ACK standard libraries. *Dispose_saved* looks for, and disposes, the named scope from the DEFAULT_TABLE.

*Restore_scope_of* allows the relinking of a previously saved scope into the symbol table. The indicated scope must have been previously saved. It is searched for in the saved scopes, and if found, is relinked as the current scope of the indicated table. The scope pointer is returned, or NULL on failure. *Restore_scope* uses the DEFAULT_TABLE.

*Current_scope_name_of* returns a copy of the name of the current scope of the indicted table. *Current_scope_name* returns a copy for the DEFAULT_TABLE.

## LOOKUP

*Look* looks up the identifier in the indicated symbol table. The lookup is started in the start scope and continues out towards the most global scope. This follows the typical block structured scope hierarchy. The search is finished when the identifier is found or when the stop scope or a closed scope is encountered.

If multiple is true (i.e. non zero) the search is commenced from where the last lookup ended, to find symbols that are multiply defined within the same scope. All lookups return a pointer to the user defined structure of the required identifier, or NULL.

*Lookup* looks up the identifier in the DEFAULT_TABLE, working out to the most global scope from the current scope, i.e. the typical lookup process.*Lookup* follows the restrictions imposed by closed scopes. *Lookup_local* looks up the identifier in the DEFAULT_TABLE, in the current scope only.

*Lookup_record* looks up the identifier name within the indicated scope. Useful for lookup within a scope not currently linked into the symbol table, i.e. for record fields that are saved within the record symbol as a scope. Returns a pointer to the user defined structure, or NULL.

*Lookup_in_scope_of* looks up the identifier only within the indicated scope in the indicated table. Returns a pointer to the user struct, or NULL. *Lookup_in_scope* searches for the identifier in the named scope, in the DEFAULT_TABLE.

*Search_saved* returns a pointer to a scope if it is in the saved scopes. Note the scope remains linked into the saved scopes. This allows (for example) iterations using *sym_iterate* to be carried out on scopes not currently in the symbol table.

*Multiple_lookup_in* allows the lookup of all symbols with the same identifier in the indicated scope. If the parameter start is non-NULL then a multiple lookup is started. For subsequent lookups on the same identifier, parameter start should be NULL. Lookups take place in the indicated scope only. Each time *multiple_lookup_in* is called the next user defined structure with the indicated identifier is returned, until there are no more of that name. NULL is then returned.

*Multiple_lookup* initiates a multiple lookup in the DEFAULT_TABLE. *Next_lookup* can be used for subsequent lookups of the identifier, in the DEFAULT_TABLE.

## SYMBOL MANIPULATION

*Insert_wild* inserts an identifier into the indicated table. The parameter wild is a user hook for iterating over selected symbols in a scope, wild values are predefined for EXPORT, IMPORT, TAME and ALL_NODES. Returns a pointer to the user defined struct, (SYMNODE_TYPE) that has been inserted into the table. The user may then copy data into the returned struct.

*Insert* uses the DEFAULT_TABLE and sets the user wildcard to the predefined value TAME.

*Insert_local* inserts the identifier into the current scope of the default table only if it does not already exist in that scope. Returns pointer to user struct, or NULL if the symbol already exists.

*Iterate_start* initialises the symbol table iterater with a scope to iterate through and the wildcard to iterate on. Predefined wildcards include EXPORT and IMPORT, these can be added to by the user as they are needed. The wildcards can be combined i.e. a wildcard of (EXPORT | IMPORT) will initialise the iterater to return all user symbols that have their wildcard set to IMPORT or EXPORT.

*Sym_iterate* uses the wild card and scope set up by the most previous *iterate_start* to return a pointer to the next user defined struct that matches the wild card. Iteration is performed by matching on the user hook within the symbol table.

Multiple wild values can be matched, all nodes can be matched by the predefined wildcard ALL_NODES. Returns a pointer to user struct, or NULL if no more symbols in the scope match the criteria.

*Reset_wild_of* resets the wildcard of the identifier if it matches the old_wild value. The wildcard is the user hook within the symbol table structure which is used by *sym_iterate* to allow selective iteration over symbols within a scope. The new value of the wildcard is new_wild. Returns true if an identifier was found that had the old wildcard value, false otherwise. *Reset_wild* looks in the current scope of the DEFAULT_TABLE

and changes the old wildcard to the new value, regardless of the old wildcard value.

*Delete_sym* searches for the symbol indicated by the pointer u_node. If it is found the symbol is deleted from the table and deallocated.

## MODULE MANIPULATION

*Import_preload_to* imports a preload file of global symbols into the current scope of the indicated table. The preload file is created using *preload_main.c,* and would typically consist of pervasive symbols such as basic language types. This is a simple method of using a saved module to load a scope. Returns true or false indicating success of the operation. *Import_preload* imports into the DEFAULT_TABLE.

*Import_module_to* uses the gen_type parameter as a wildcard to import symbols from the indicated module (scope) into the current scope of the specified table. The module is first looked for in the saved scopes (if they are being saved) then on secondary storage. *Import_module_to* returns the current scope pointer, NULL if unsuccessful or no matching symbols found.

All of the following import routines have aliases which use the predefined wildcard EXPORT. These show how features such as modules could be handled using the symbol table library. *Import_module* imports all symbols in the named module that have a wildcard value of EXPORT.

*Import_from_module_to* is the same as *import_module_to* except that only the symbol with the specified identifier, and wildcard matching gen_type, is loaded from the module. Returns NULL if unsuccessful. *Import_from_module* imports the specified symbol from the module if the wildcard is EXPORT.

*Import_generics_to* is the same as *import_module_to* except that symbols matching the wildcard gen_type are loaded from the indicated scope. *Import_exports* imports all symbols with wildcard value EXPORT from the indicated scope into the current scope of the DEFAULT_TABLE.

*Import_from_generics_to* is the same as *import_from_module_to* except that the identifier is loaded from the indicated scope rather than a named module. *Import_from_exports* imports the symbol with the specified identifier from the indicated scope if it's wildcard value is EXPORT.

## DEBUGGING

*Dump_table* is the general symbol table debugging tool, the symbol table is dumped to the indicated file. The flag indicates whether the symbol table (TABLE_NODES) or the saved scopes (SAVED_NODES) are to be dumped.

*Snap_shot* is similar to *dump_table* but takes advantage of the fact that many table dumps will take the same parameters as the previous call. *Snap_shot* dumps the symbol table, automatically taking the same parameters as the previous call to *dump_table*. The filename's are numbered sequentially, snapshot.1 etc. The initial settings are to dump the table nodes of DEFAULT_TABLE.

## USAGE

The symbol table library provides a range of routines to supply the user with a symbol table useable for a wide range of programming languages. The library is intended to hide from users the underlying table structure, the routines providing an interface with this structure.

This is not a true library as the user must compile the library, specifying their own parameters. This produces a purpose built object file which may be linked into a compiler.

The library is designed for use within compilers of typical block structured languages, for example *C, Pascal,* and *Modula2*. It provides all common symbol table operations along with extensions for languages which contain explicit scope control.

These extensions include module facilities, saving of scopes and symbol tables, open and closed scopes, restricted scope lookup and multiple lookups within specified scopes. The library routines operate on a hash based symbol table structure, the parameters of which are user specified. The basic sequence of events required to use the library are as follows:

1. Make copies of the files *user_def.h* and *user_code.c*. The first of these files details the user parameters that may be selected. This allows the symbol table to be tailored to individual requirements. The struct definition, which will contain user data within the symbol table, must also be specified. The second file gives a guideline for the user about routines that are required for saving and loading scopes.

2. The user copy of *user_def.h* must then be #included within the library so that it can be compiled with the user specifications. This is done by altering the file *user_def_filename* to indicate the full pathname of the users copy of *user_def_.h*.

3. The file *symlib.c* can then be compiled, by using the supplied makefile, to an object file which may be linked into a compiler.

4. Any files which use the symbol table routines must #include the header file *sym_type.h*.

## EXAMPLES

Two subdirectories */symlib/Modula* and */symlib/Pascal* contain demonstration parsers which use this symbol table library. These use *yacc* and *lex* generated parsers and scanners coupled with the symbol table library to do some elementary semantic checking. (The Modula parser is very loosely based on Modula2, it is basically the Pascal parser with modules). The directories include some simple example programs/ modules.

## FILES

The symbol table library is in the directory

*EM/modules/symlib*

The library files

*symlib/sym_type.h*
*symlib/user_def.h*
*symlib/extern.h*
*symlib/alias.h*
*symlib/user_def_filename*
*symlib/user_code.c*
*symlib/primitives.c*
*symlib/util.c*
*symlib/sym_io.c*
*symlib/import.c*

*symlib/hash_fn.c*
*symlib/sym_debug.c*
*symlib/preload_main.c*
*symlib/symlib.c*
*symlib/Makefile*
*symlib/makefile_template*

The example directories
*symlib/Modula*
*symlib/Pascal*

## BUGS

Hmmmmmmmmmm... There is no way of stopping potential users from trying to directly manipulate the underlying symbol table structure, rather than using the supplied routines. Problems could occur if users delete certain of the #defined constants in their copy of *user_def.h*. These constants are clearly indicated.

## SEE ALSO

ack(1), yacc(1), lex(1)

## AUTHOR

Warwick Heath, University of Canterbury, New Zealand.

# Appendix B.

## Library file structure.

The ACK environment is C under Unix. C has extensive separate compilation facilities and promotes the use of many separate source files. These files are linked together to form a runfile. A library is an object file which may be linked into a number of different runfiles. As such, my routines are not a true library because the user must compile the library before linking it into their runfile. This is because there are a number of user parameters which can be altered, depending on symbol table requirements.

The symbol table library is in the directory **EM/modules/symlib**. Online help can be obtained by '**man symlib**'. The files comprising my project are split between code files and definition, or in C terminology, header files. During development the header files were physically included within the separately compiled code files at compile time. At the end of the development stage all files were combined into a single compilation unit called **symlib.c** so that users of the library only have to compile and link one file into their compiler. The makefile in the **symlib** directory can be used to compile **symlib.c** into the required object file.

Following is a synopsis of the files in my project and their function.

**User_def.h**

This header file contains all the user controllable parameters, for example the size of the symbol table. This file must be included within all files that use symbol table routines. Various other files also must be included in all files using the symbol table routines. All of these files are combined by **sym_type.h** which is then the only file the user has to include within their files.

Besides various parameters, the **user_def.h** file also contains the definition of the user's C structure that is to be stored in the symbol table. All symbol table routines passing data back to the user pass a pointer to an instance of this user defined structure. **User_def.h** also contains default pathnames for module input and output and defines the names of the user's module input output routines. The user has their own copy of this file.

**User_code.c**

This file contains a template for the module input and output routines that the user must supply if modules are being used. This is a template so it is not necessary to have this actual file, the routines just must be included somewhere in the user's code.

**User_def_filename**

This file contains the file and pathname of the users customised copy of **user_def.h**. Therefore before compilation the user must change this file to specify their copy of **user_def.h.**

**Sym_type.h**

This contains the declarations for the symbol table structure and the memory allocation routines for these structures. The **user_def.h** file is included, as the parameters specify the size of the symbol table and whether or not the user is saving scopes etc. This file is included in all symbol table library files which is why they must be recompiled for each different user. The user must also include this file within any files that use the symbol table.

**Extern.h**

This file contains the C external declarations of all user accessable symbol table routines. This file is included in **sym_type.h. Extern.h** gives comprehensive documentation of these routines, detailing the parameter and return types of all routines. This information is duplicated in the manual page.

**Alias.h**

In many cases the user may wish to access routines using their default table and the current scope of that default table. Some alias's for common occurrences of this are defined in **alias.h** as is documentation of these routines. This file is also included within **sym_type.h.**

The other symbol table library files are code files containing the symbol table manipulation routines. These are **util.c, import.c, sym_debug.c, sym_io.c** and **primitives.c.**

# Appendix C.

## Modula2 test parser example.

This appendix lists a simple program used to test the Modula parser that was developed to verify the symbol table library. The accompanying notes give a short description of the functions that must be performed as the modules are parsed. Comments in the code detail the visible symbols in each module block.

```
DEFINITION MODULE subsetOne;          (* Note 1 *)

FROM Global IMPORT char, integer;     (* Note 2 *)

     CONST
          OneOne = 1;
          OneTwo = 2;
     TYPE
          OneType = array[1..OneTwo] of integer;
     VAR
          OneVar : char;

END subsetOne.
```

```
DEFINITION MODULE subsetTwo;          (* Note 3 *)

FROM subsetOne IMPORT OneOne, Onetype;
FROM Global IMPORT integer;

     CONST
          TwoOne = OneOne;
     VAR
          TwoVar : OneType;

     FUNCTION TwoFunc: integer;       (* Note 4 *)

END subsetTwo.
```

```
(* Main program unit *)
MODULE mainprogram;                          (* Note 5 *)

IMPORT subsetOne;                            (* Note 6 *)
FROM Global IMPORT write, integer, char;
FROM subsetTwo IMPORT TwoFunc;

     VAR
          MainVar : integer;

     MODULE nested;                          (* Note 7 *)

     EXPORT NestOne, NestTwo, NestNestOne;          (* Note 8 *)

     CONST
          NestOne = 1;
          NestTwo = 2;

          MODULE nestedAgain;                (* Note 9 *)
          EXPORT NestNestOne;

          CONST
               NestNestOne = 1;
          BEGIN
          (* NestNestOne is visible *)
          END nestedAgain;                   (* Note 10 *)

     BEGIN
     (* NestOne, NestTwo and NestNestOne are visible *)
     END nested;                             (* Note 11 *)

BEGIN
     (*   NestOne, NestTwo, NestNestOne,
          write, integer, char,
          TwoFunc, OneOne, OneTwo,
          OneType, MainVar and OneVar are visible *)

     MainVar := NestNestOne;                 (* Note 12 *)
     MainVar := TwoFunc;
     MainVar := NestOne;
     MainVar := NestTwo;
     MainVar := OneOne;
     OneVar := 'C';
     write(OneVar);
END mainprogram.
```

## NOTES:

1. This is the first of the two separately parsed definition modules. In Modula2 this would have an analogous implementation module. Every symbol defined in the definition module is an exported symbol. In earlier versions of Modula this was explicitly stated by using the reserved word EXPORT, but this has now been discarded. As each symbol is parsed, it is inserted into the module as an export. At the end of parsing this module the scope is saved to secondary storage.

2. The test Modula parser has been implemented in a fairly sparse manner, there are no built in types or functions such as integer, char or abs(). This was merely a design decision, it would be a simple task to preload the symbol table with pervasive identifiers as was done in the Pascal parser. As a result of this decision, all language types and functions that are used in a module must be imported from Global which is the 'system' module. These are inserted into the module but care must be taken not to save them as part of the scope when the module is exited.

3. This is the second of the separately parsed modules. Note that this module must be parsed after the first module as this module imports some symbols from the first module. The test parser would halt with an error message, indicating that the module subsetOne could not be found to import from, if the modules were parsed in the wrong order.

4. As this Modula parser was converted from the Pascal parser it contains some obvious Pascal syntax, the use of FUNCTION rather than a procedure which returns a value, is one of these. This function is an exported symbol, along with the constant and variable also in this module.

5. This is the program module, analogous to main() in the C programming language. A closed scope is created.

6. The main module has multiple imports, including the importation of all the exported symbols from module subsetOne. Note that the procedure write must be imported from the Global module.

7. A local module. This is a closed scope, only exports and imports may cross the scope boundaries, in this case there are no imports. The exports must be forward declared at this stage, so that any undefined exported symbols can be trapped when the module exits.

8. When the constant declarations for NestOne and NestTwo are encountered the identifiers that have been forward declared are checked. In this case the information from the constant declarations will be used to fill in the information for the exported symbols.

9. Another nested module, another closed scope is created. Again the constant declaration is matched with the export declaration.

10. At this point all symbols in the scope are saved and the scope delinked from the table.

11. The scope from the exited module is now iterated through, all symbols that were declared as exports are matched, and are inserted into the current module. The symbol NestNestOne is therefore imported, and it's forward declaration is searched for. If a forward declaration is not found then NestNestOne would be changed from export to a normal symbol, i.e. it would have reached the limit of it's exportation. However in this case NestNestOne is exported again, so the forward declaration is merely deleted, and NestNestOne is left as an export.
The scope of module nested is then closed.

12. The main program module at this point iterates through the saved scope from module nested, importing all symbols flagged as exports, i.e. NestNestOne, NestOne, and NestTwo. These symbols are then used within the main program body, along with the symbols that were initially imported in the import clause at the start of the program.